



895.06

Рейтинг

OTUS

Цифровые навыки от ведущих экспертов



badcasedaily1 2 ноя 2023 в 18:37

concurrent.futures в Python



Простой



11 мин



14K

Блог компании OTUS, Python*, Программирование*

Обзор

Подписаться

ИНФОРМАЦИЯ

Сайт

otus.ru

Дата регистрации

22 марта 2017

Дата основания

1 апреля 2017

Численность

101–200 человек

Местоположение

Россия

Представитель

OTUS

ВИДЖЕТ





Привет, Хабр! Сегодня мы взглянем на одну из самых интересных библиотек в Python для работы с параллельным выполнением задач - **concurrent.futures**.

↑ +17 ↓ 86 ↗ 3

параллельно. Это может быть I/O-операции, которые блокируют основной поток, или вычисления, требующие большого объема процессорных ресурсов. Здесь на помощь приходит `concurrent.futures` - модуль, предоставляющий высокоуровневый интерфейс для асинхронного и параллельного выполнения задач.

Какие преимущества предоставляет этот модуль?

1. **Простота использования:** `Concurrent.futures` предоставляет простой и интуитивно понятный API для запуска задач параллельно. Это позволяет сосредоточиться на

В КОНТАКТЕ

решении задачи, а не на деталях многозадачности.

2. **Автоматическое масштабирование:** Модуль позволяет легко масштабировать задачи, выполняемые в пулах потоков (ThreadPoolExecutor) и пулах процессов (ProcessPoolExecutor). Вы можете использовать их в зависимости от характера задачи и доступных ресурсов.
3. **Удобная обработка результатов:** Concurrent.futures предоставляет Future объекты, которые позволяют отслеживать выполнение задач и получать результаты, когда они готовы.
4. **Отсутствие необходимости заботиться о GIL:** В отличие от многих других способов параллельного выполнения в Python, concurrent.futures позволяет избежать проблем, связанных с Global Interpreter Lock (GIL), что делает его отличным выбором для многозадачных приложений.



Основы concurrent.futures

Асинхронность и параллелизм – два важных понятия, которые позволяют нам оптимизировать выполнение задач в наших программах:

1. **Асинхронность:** Этот подход позволяет программе продолжать выполнение других задач, не блокируя основной поток выполнения. Он часто используется для обработки I/O-операций, которые могут занимать значительное время, например, чтение данных из сети или файловой системы. Асинхронный код обычно использует событийную модель и обратные вызовы (callbacks) для уведомления о завершении операции.
2. **Параллелизм:** В то время как асинхронность позволяет выполнять задачи, не блокируя основной поток, параллелизм предоставляет возможность выполнения нескольких задач одновременно. При этом каждая задача может выполняться в собственном потоке или процессе, в зависимости от используемого механизма.

БЛОГ НА ХАБРЕ

7 часов назад

Препарируем Wazuh. Часть 3:
источники не из коробки

 354  1 +1

14 часов назад

Клеточная архитектура

 1.2K  2 +2

20 апр в 21:10

Кратко про Serde в Rust

1. **ThreadPoolExecutor:** ThreadPoolExecutor – это класс, предоставляющий пул потоков для выполнения задач. Этот инструмент особенно полезен, когда задачи связаны с I/O-операциями, которые блокируют поток выполнения. Пример использования:

```
from concurrent.futures import ThreadPoolExecutor

def task_function(param):
    # Реализация задачи
    return result

# Создание ThreadPoolExecutor с 4 рабочими потоками
with ThreadPoolExecutor(max_workers=4) as executor:
    # Запуск задачи асинхронно
    future = executor.submit(task_function, param)
    # Ожидание результата и получение его
    result = future.result()
```

2. **ProcessPoolExecutor:** ProcessPoolExecutor предоставляет пул процессов для выполнения задач. Этот инструмент полезен, когда задачи могут выполняться независимо и имеют высокую вычислительную нагрузку:

```
from concurrent.futures import ProcessPoolExecutor

def task_function(param):
    # Реализация задачи
    return result
```

 1.3K  4 +4

20 апр в 12:11

SaltStack: управление конфигурациями

 1.2K  4 +4

19 апр в 23:05

Пять лучших NLP инструментов для работы с русским языком на Python

 4.9K  3 +3

```
# Создание ProcessPoolExecutor с 4 рабочими процессами
with ProcessPoolExecutor(max_workers=4) as executor:
    # Запуск задачи параллельно
    future = executor.submit(task_function, param)
    # Ожидание результата и получение его
    result = future.result()
```

Для создания экземпляра ThreadPoolExecutor или ProcessPoolExecutor, вы можете установить максимальное количество рабочих потоков или процессов с помощью параметра `max_workers` . Это позволяет вам настроить параллельное выполнение под ваше конкретное приложение:

```
from concurrent.futures import ThreadPoolExecutor, ProcessPoolExecutor

# Создание ThreadPoolExecutor с 8 рабочими потоками
thread_executor = ThreadPoolExecutor(max_workers=8)

# Создание ProcessPoolExecutor с 4 рабочими процессами
process_executor = ProcessPoolExecutor(max_workers=4)
```

Future объекты представляют асинхронные задачи, которые были отправлены на выполнение в пуле потоков или процессов. Они предоставляют удобный способ отслеживания статуса и получения результатов задачи:

```
from concurrent.futures import ThreadPoolExecutor

def task_function(param):
    # Реализация задачи
```

```
    return result

with ThreadPoolExecutor(max_workers=4) as executor:
    future = executor.submit(task_function, param)

# Ожидание результата
result = future.result()
```

Future объекты также позволяют обработать ошибки, которые могли возникнуть при выполнении задачи:

```
try:
    result = future.result()
except Exception as e:
    print(f"Произошла ошибка: {e}")
```

Задачи и Пулы потоков (Thread Pools)

Задачи - это базовые строительные блоки параллельного выполнения. Они могут быть представлены в виде функций, которые выполняют какую-либо работу. Например, задача может быть функцией, которая загружает данные из сети, анализирует текст или обрабатывает изображение:

```
def load_data_from_server():
    # Загрузка данных из сети
    pass
```



```
def process_text(data):  
    # Анализ и обработка текста  
    pass  
  
def process_image(image):  
    # Обработка изображения  
    pass
```

ThreadPoolExecutor предоставляет пул потоков для выполнения задач. Он работает следующим образом: вы создаете экземпляр ThreadPoolExecutor, указывая количество рабочих потоков, и отправляете задачи на выполнение в пуле.

```
from concurrent.futures import ThreadPoolExecutor  
  
# Создание ThreadPoolExecutor с 4 рабочими потоками  
with ThreadPoolExecutor(max_workers=4) as executor:  
    # Запуск задач асинхронно  
    future1 = executor.submit(load_data_from_server)  
    future2 = executor.submit(process_text, data)  
    future3 = executor.submit(process_image, image)  
  
# Ожидание результата и получение его  
result1 = future1.result()  
result2 = future2.result()  
result3 = future3.result()
```

ThreadPoolExecutor автоматически управляет пулом потоков и распределяет задачи между рабочими потоками, что делает его удобным инструментом для асинхронного выполнения задач.

ThreadPoolExecutor предоставляет методы для управления пулом потоков. Например, метод `shutdown()` позволяет корректно завершить работу пула после завершения всех задач:

```
from concurrent.futures import ThreadPoolExecutor

with ThreadPoolExecutor(max_workers=4) as executor:
    # Запуск задач
    ...

# После завершения всех задач, пул потоков автоматически закрывается
```

Определение правильного количества рабочих потоков чрезвычайно важно. Слишком много потоков может привести к избыточным накладным расходам, а слишком мало - к неэффективности параллельного выполнения. ThreadPoolExecutor позволяет легко управлять этим параметром с помощью `max_workers`.

```
from concurrent.futures import ThreadPoolExecutor

# Создание ThreadPoolExecutor с динамическим управлением рабочими потоками
with ThreadPoolExecutor(max_workers=None) as executor:
    # Он автоматически адаптирует количество потоков к нагрузке
    ...
```

Рассмотрим различные сценарии использования ThreadPoolExecutor: параллельная загрузка данных из сети, обработка больших объемов текста и многозадачная обработка изображений:

Параллельная загрузка данных из сети

```
from concurrent.futures import ThreadPoolExecutor
import requests

def fetch_url(url):
    response = requests.get(url)
    return response.content

urls = ["http://example.com", "http://example.org", "http://example.net"]

with ThreadPoolExecutor(max_workers=4) as executor:
    results = list(executor.map(fetch_url, urls))
```

Обработка больших объемов текста

```
from concurrent.futures import ThreadPoolExecutor

def process_text(text):
    # Реализация обработки текста
    pass

text_data = [...]

with ThreadPoolExecutor(max_workers=4) as executor:
    results = list(executor.map(process_text, text_data))
```

Многозадачная обработка изображений

```
from concurrent.futures import ThreadPoolExecutor
from PIL import Image

def process_image(image_path):
    image = Image.open(image_path)
    # Реализация обработки изображения
    pass

image_paths = [...]

with ThreadPoolExecutor(max_workers=4) as executor:
    results = list(executor.map(process_image, image_paths))
```

Многозадачность и Пулы процессов (Process Pools)

ProcessPoolExecutor предоставляет пул процессов, что позволяет выполнить задачи параллельно в отдельных процессах. Это особенно полезно, когда у вас есть задачи, которые могут выполняться независимо друг от друга, и вы хотите максимально использовать ресурсы многопроцессорных систем.

Пример использования ProcessPoolExecutor:

```
from concurrent.futures import ProcessPoolExecutor

def task_function(param):
    # Реализация задачи
    return result

# Создание ProcessPoolExecutor с 4 рабочими процессами
```

```
with ProcessPoolExecutor(max_workers=4) as executor:  
    # Запуск задачи параллельно  
    future = executor.submit(task_function, param)  
    # Ожидание результата и получение его  
    result = future.result()
```

В этом примере мы создаем пул процессов с помощью `ProcessPoolExecutor`, отправляем задачу на выполнение и затем ожидаем ее завершения. `ProcessPoolExecutor` автоматически управляет процессами, что делает его мощным инструментом для параллельного выполнения задач.

Различия между `ThreadPoolExecutor` и `ProcessPoolExecutor`:

1. **Потоки vs. Процессы:** `ThreadPoolExecutor` использует потоки, работающие в пределах одного процесса, в то время как `ProcessPoolExecutor` использует отдельные процессы для каждой задачи. Это позволяет `ProcessPoolExecutor` избегать проблем с Global Interpreter Lock (GIL), которые могут возникнуть в `ThreadPoolExecutor`.
2. **Ресурсы и производительность:** `ThreadPoolExecutor` обычно требует меньше системных ресурсов, так как потоки делят один и тот же адресное пространство процесса. Однако `ProcessPoolExecutor` может обеспечить более высокую производительность в случае многозадачных вычислений, так как каждая задача выполняется в отдельном процессе.
3. **Сериализация данных:** В `ProcessPoolExecutor` данные, передаваемые между процессами, должны быть сериализованы и десериализованы. Это может потребовать дополнительных усилий и влиять на производительность. В `ThreadPoolExecutor` такой необходимости нет.

Передача данных между процессами в `ProcessPoolExecutor` требует сериализации и десериализации объектов. Это означает, что данные, передаваемые в задачу и

возвращаемые из нее, должны быть сериализуемыми.

Пример передачи данных между процессами:

```
from concurrent.futures import ProcessPoolExecutor

def process_data(data):
    # Реализация обработки данных
    pass

data_list = [...]

with ProcessPoolExecutor(max_workers=4) as executor:
    results = list(executor.map(process_data, data_list))
```

Здесь `data_list` передается в задачу, и результаты обработки данных возвращаются в основной процесс.

Рассмотрим пару примеров применения вышесказанного:

Вычисление суммы квадратов чисел в параллельных процессах

```
from concurrent.futures import ProcessPoolExecutor

def square_and_sum(numbers):
    return sum(x ** 2 for x in numbers)

numbers_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
with ProcessPoolExecutor(max_workers=4) as executor:  
    results = list(executor.map(square_and_sum, [numbers_list] * 4))
```

Обработка изображений в параллельных процессах

```
from concurrent.futures import ProcessPoolExecutor  
from PIL import Image  
  
def process_image(image_path):  
    image = Image.open(image_path)  
    # Реализация обработки изображения  
    pass  
  
image_paths = [...]  
  
with ProcessPoolExecutor(max_workers=4) as executor:  
    results = list(executor.map(process_image, image_paths))
```

Параллельное вычисление матричного произведения

```
from concurrent.futures import ProcessPoolExecutor  
import numpy as np  
  
def matrix_multiply(matrices):  
    A, B = matrices  
    result = np.dot(A, B)  
    return result
```

```
matrix_A = np.random.rand(100, 100)
matrix_B = np.random.rand(100, 100)

with ProcessPoolExecutor(max_workers=4) as executor:
    results = list(executor.map(matrix_multiply, [(matrix_A, matrix_B)] * 4))
```

Очереди и многозадачность

Очереди позволяют распределять задачи между разными рабочими процессами или потоками, а **многозадачность** обеспечивает эффективное выполнение этих задач.

Основные преимущества применения очередей и многозадачности:

1. **Увеличение производительности:** Многозадачность позволяет распараллеливать выполнение задач, что может значительно повысить производительность вашего приложения.
2. **Управление ресурсами:** Вы можете эффективно управлять ресурсами вашей системы, распределяя задачи на выполнение в соответствии с их приоритетом и доступными ресурсами.
3. **Обработка больших объемов данных:** Очереди и многозадачность идеально подходят для обработки больших объемов данных, так как они позволяют эффективно разделить задачи между несколькими потоками или процессами.

Producer-consumer паттерн - это популярный способ реализации очередей и многозадачности. В этом паттерне задачи генерируются "производителями" и помещаются в очередь, откуда их забирают "потребители" для выполнения. `Concurrent.futures` предоставляет удобные средства для реализации этого паттерна.

Рассмотрим три примера кода, демонстрирующих применение producer-consumer паттерна с использованием concurrent.futures:

Многозадачная загрузка изображений из сети

```
from concurrent.futures import ThreadPoolExecutor
import requests

def download_image(url):
    response = requests.get(url)
    return response.content

image_urls = ["url1.jpg", "url2.jpg", "url3.jpg"]

with ThreadPoolExecutor(max_workers=4) as executor:
    image_futures = {executor.submit(download_image, url): url for url in image_urls}

for future in concurrent.futures.as_completed(image_futures):
    url = image_futures[future]
    image_data = future.result()
    # Обработка загруженного изображения
```

В этом примере, задачи "производителями" являются запросы на загрузку изображений, а задачи "потребителями" - обработка полученных данных.

Обработка данных с использованием многозадачности

```
from concurrent.futures import ProcessPoolExecutor
```



```

data_to_process = [...]

def process_data(data):
    # Обработка данных
    pass

with ProcessPoolExecutor(max_workers=4) as executor:
    results = list(executor.map(process_data, data_to_process))

```

В этом примере, "производителем" является процесс, создающий данные для обработки, а "потребителем" - процессы, выполняющие фактическую обработку.

Параллельное выполнение вычислений

```

from concurrent.futures import ThreadPoolExecutor

def perform_computation(data):
    # Вычисления
    pass

computation_data = [...]

with ThreadPoolExecutor(max_workers=4) as executor:
    results = list(executor.map(perform_computation, computation_data))

```

В этом примере, "производителем" могут быть данные, которые требуют вычислений, а "потребителем" - потоки, выполняющие эти вычисления.

Отслеживание выполнения и обработка результатов

Future объекты предоставляют способ отслеживать выполнение асинхронных задач и получать их результаты, позволяют вам мониторить статус и результаты задач.

Пример использования Future объектов:

```
from concurrent.futures import ThreadPoolExecutor

def task_function(param):
    # Реализация задачи
    return result

with ThreadPoolExecutor(max_workers=4) as executor:
    future = executor.submit(task_function, param)

# Отслеживание выполнения задачи
if future.done():
    result = future.result()
else:
    # Задача все еще выполняется
```

Мы создаем Future объект, представляющий задачу, и затем отслеживаем его выполнение с помощью методов `done()` и `result()`. Мы можем узнать, завершилась ли задача, и получить ее результат, когда она завершится.

При разработке приложений важно обрабатывать исключения и ошибки, которые могут возникнуть во время выполнения задач. Future объекты предоставляют механизм для обработки исключений, возникающих в задачах:

```
from concurrent.futures import ThreadPoolExecutor

def task_function(param):
    try:
        # Реализация задачи
        result = perform_task(param)
    except Exception as e:
        # Обработка исключения
        result = handle_exception(e)
    return result

with ThreadPoolExecutor(max_workers=4) as executor:
    future = executor.submit(task_function, param)

# Обработка исключения, если оно произошло
try:
    result = future.result()
except Exception as e:
    handle_exception(e)
```

В этом примере, мы внутри задачи "ловим" исключение, если оно возникает, и затем обрабатываем его. Затем, при получении результата с Future объекта, мы также обрабатываем исключение, если оно было выброшено внутри задачи.

В некоторых сценариях вам может потребоваться собрать и агрегировать результаты выполнения нескольких задач. Concurrent.futures предоставляет удобные способы сделать это:

```
from concurrent.futures import ThreadPoolExecutor

def task_function(param):
    # Реализация задачи
    return result

params = [param1, param2, param3]

with ThreadPoolExecutor(max_workers=4) as executor:
    futures = [executor.submit(task_function, param) for param in params]

# Сбор результатов выполнения задач
results = [future.result() for future in futures]

# Агрегация результатов
aggregated_result = aggregate_results(results)
```

В этом примере мы создаем Future объекты для каждой задачи и затем собираем результаты выполнения задач в список. Затем эти результаты могут быть агрегированы по вашим потребностям.

Использование concurrent.futures позволяет создавать быстрые и эффективные приложения, эффективно управлять задачами и избегать проблем с Global Interpreter Lock (GIL).

Статья подготовлена в преддверии старта специализации Системный аналитик. Узнать подробнее о специализации можно [тут](#).

Теги: otus, python

Хабы: Блог компании OTUS, Python, Программирование

Редакторский дайджест



Присылаем лучшие статьи раз в месяц



OTUS

Цифровые навыки от ведущих экспертов

[Сайт](#) [ВКонтакте](#) [Telegram](#)



36

Карма

220

Рейтинг



Подписаться

artem @badcasedaily1

Пользователь



Комментарии 3

Публикации



atomlib 18 часов назад

48 лет вместе с Zilog Z80

Простой 10 мин 11K

+95 32 25 +25



Shkaff 9 часов назад

На гребне гравитационной волны: космический детектор LISA

Средний 12 мин 1.8K

+38 12 11 +11



Bright_Translate 13 часов назад

Поиск по коду — это сложно

Простой 5 мин 3K

Обзор

Перевод

+31 18 1 +1



neskuchan 10 часов назад

Найти работу в IT: миссия 2024

Простой 7 мин 3.1K

Обзор

♦ +26

📖 17

💬 6 +6



Grigory_Otrepyev 9 часов назад

Запуск ракеты Ангара — неделю спустя

🔴 Сложный

🕒 6 мин

👁 9.1K

Мнение

♦ +23

📖 9

💬 20 +20



kkxen 15 часов назад

Как запустить IT-подкаст: прошли этот путь и расскажем обо всех подводных камнях

🕒 7 мин

👁 2K

Кейс

♦ +22

📖 16

💬 2 +2



dalerank 4 часа назад

Я скучаю по механикам из старых игр

🕒 13 мин

👁 3.1K

♦ +21

📖 11

💬 42 +42



MarkNest 11 часов назад

Стала доступнее веб-страничка, которая строит спектр отражения и пропускания света слоистой средой

🕒 2 мин 👁 1.3K

💎 +15 📖 13 💬 2 +2



ibessonov 6 часов назад

Миллер, Рабин, вектор

🔒 Сложный 🕒 16 мин 👁 977

💎 +13 📖 15 💬 6 +6



GlobalSign_admin 5 часов назад

Фальшивые криптокошельки в официальном каталоге Ubuntu — индикатор более серьёзной угрозы

🕒 3 мин 👁 1.6K

💎 +12 📖 6 💬 6 +6

Показать еще

Ваш аккаунт

- Войти
- Регистрация

Разделы

- Статьи
- Новости
- Хабы
- Компании
- Авторы
- Песочница

Информация

- Устройство сайта
- Для авторов
- Для компаний
- Документы
- Соглашение
- Конфиденциальность

Услуги

- Корпоративный блог
- Медийная реклама
- Нативные проекты
- Образовательные программы
- Стартапам

