Boosting REST API Performance Using Asyncio

Johannes Krohn

Technical Marketing Engineer

10. Sep 2019

Agenda

Code, examples available at https://github.com/jeokrohn/duasync

- Python Performance
- Concurrency, ...
- concurrent.futures
- Asyncio
- Summary

Python Performance

Python Performance

- Depending on the workload Python code can be slower than other language like for example Java
- A lot of workloads are not really CPU bound
  - Other factors: network, queries, ...
  - Faster running code does not really help
- Python execution speed does not really matter when working on I/O bound problems

Demo – Scraping “Spiegel” sequentially

I/O is very slow

- Each REST call takes ”forever”
  - Creating the space
  - Adding a user to the space
- Python code the majority of the time just idles and needs to wait for a response

Concurrency, ...

Practical Example: Chess Exhibition (serial)

- Assumption:
  - 24 opponents
  - Master moves in 5 seconds
  - Other players move in 55 seconds
  - Game averages at 30 move pairs
- Each game runs 30 minutes
- 24 sequential games: 12 hours

Asynchronous Python for the Complete Beginner, Miguel Grinberg, Pycon 2017: https://www.youtube.com/watch?v=iG6fr81xHKA

Practical Example: Chess Exhibition (async)

- Master moves on first game
- Then moves to 2nd, 3rd, 4th, ..
- A move on all 24 games takes the master: 24 x 5 sec = 2 min
- After two minutes the 1st game is again ready for her move
- 24 games are completed in 2 min x 30 = 1 h

Asynchronous Python for the Complete Beginner, Miguel Grinberg, Pycon 2017: https://www.youtube.com/watch?v=iG6fr81xHKA

Parallelism, Concurrency, ...

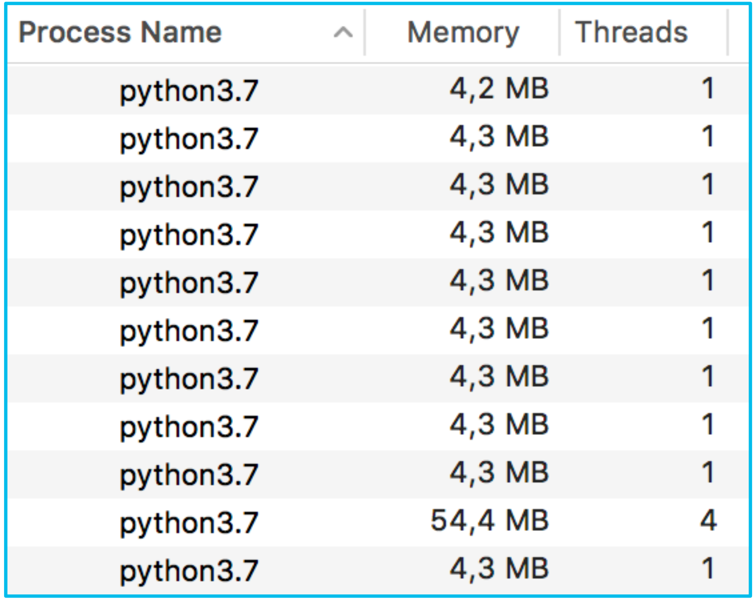
- Parallelism: performing multiple operations at (virtually) the same time
  - Multiprocessing: spreading tasks over CPUs/cores
  - Good for CPU bound tasks
- Concurrency: multiple tasks can run in an overlapping manner
  - Does not imply parallelism
- Threading: concurrent execution model, threads take turns
  - Better for I/O bound tasks
- Python package: `concurrent.futures`

Demo – Scraping “Spiegel”

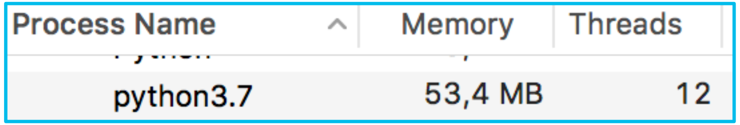
`concurrent.futures`

Findings: `concurrent.futures`

- ProcessPoolExecutor:
  - One Python process per task
  - Single thread per process
  - More memory intensive
- ThreadPoolExecutor:
  - Single Python process for all tasks
  - One thread per task



Process Name	Memory	Threads
python3.7	4,2 MB	1
python3.7	4,3 MB	1
python3.7	4,3 MB	1
python3.7	4,3 MB	1
python3.7	4,3 MB	1
python3.7	4,3 MB	1
python3.7	4,3 MB	1
python3.7	4,3 MB	1
python3.7	4,3 MB	1
python3.7	54,4 MB	4
python3.7	4,3 MB	1



Process Name	Memory	Threads
python3.7	53,4 MB	12

Asyncio Programming

- Single-threaded, single-process
- Cooperative multitasking
- “feeling of concurrency”
- What does ”asynchronous” mean?
  - Async code can “pause” (wait for a result or events) and let other code run
  - Quasi concurrent execution of multiple tasks
  - ”Cooperative”: execution is passed on when one task pauses

Asyncio Programming in Python

- Suspend and Resume
- Event loop: keep track of all async functions & state; schedule execution
- Options
  - Callbacks
  - Generators or coroutine functions
  - Async/await
- Options: Asyncio, Twisted, Tornado, …

Blocking Library Functions

- Blocking calls are incompatible w/ async programming
  - socket.*, select.*
  - threading.*
- Async frameworks like (asyncio) need to provide replacements for these
- Fallback: run sync code in separate thread or process pool (`concurrent.futures`)
- No async support for file I/O

Asyncio

Asyncio: One Way for Async Python Programming

- Coroutines: `async` functions
- `await`: wait for result (pause) → execution can be passed to other coroutine
- Event loop: keep track of coroutines, state, scheduling
- Schedule tasks:
  - `loop.create_task()`
  - `asyncio.ensure_future()`
  - `asyncio.create_task()` – Python 3.7+ only

Demos: async1, async2, async3

aiohttp: Async Web Clients

- asyncio web client
- Replacement for synchronous `requests` library
- High-Level: aiohttp.`ClientSession`() object
  - Connection pooling
  - Cookie jar
  - Keepalive
  - …
- Low-Level: aiohttp.request()

Demo: Scraping “Spiegel” w/ asyncio

Finding: `asyncio/aiohttp`

- Creating all tasks at the same time:
  - HTTP GET requests sent for all URLs
  - Issues:
    - Servicing the responses takes too long → server timeouts
    - Too many open files
  - Need to throttle the requests (similar to `max_workers` with `concurrent.futures`
  - For example: semaphore

Applying `asyncio` to REST APIs

- Using `aiohttp` enables asynchronous REST calls
- Simply need to create the required calls using `aiohttp` methods

Demo: Webex Teams demos (sync vs async)

Limitation of “Naive” REST API Approach

- No readily available replacement for sync libraries
  - Each endpoint has to be coded explicitly
- Webex Teams specific:
  - Support for 429 rate limiting
  - Pagination
  - ...
- Potential solution: use ”non asyncio” library and `concurrent.futures` with `ThreadPoolExecutor`

Summary

What Form of Concurrency is “best”?

```
if io_bound:
    if io_very_slow or lots_of_connections:
        print(‘use asyncio’)
    else:
        print(‘use threads’)
else:
    print(‘use processes’)
```

Problems with asyncio

- Need to think “asyncio”
- Hard to migrate existing code
- Some sync libraries don’t have async equivalence (for example `webexteamssdk`)

Other Services

- FTP (server/client): aioftp
- Timeouts: async-timeout
- SSH: asyncssh
- Client WebSockets: aiohttp
- Interaction with network devices: netdev
- Others: https://github.com/aio-libs

References

- https://realpython.com/async-io-python/
- https://realpython.com/python-concurrency/
- https://training.talkpython.fm/courses/explore_async_python/async-in-python-with-threading-and-multiprocessing
- https://docs.python.org/3/library/asyncio.html
- https://docs.python.org/3/library/concurrent.futures.html