Event loop

Friday, 20 September 2024 6:15 PM

asyncio → get_event

loop = asyncio.get_running_loop() ↑ better than get event

loop.call_soon(fn)

2.a asyncio has a debug mode

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

or

python3 -X dev program.py

or

env vars → PYTHONASYNCIODEBUG=1

Can help see task times

default warns if call is more than 100ms

can change loop.slow_callback_duration

---

chap 2 Summary

asyncio.run() execute single coroutine

chap 3

Build cmd line echo server

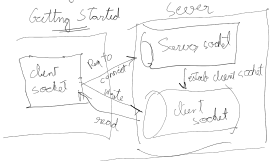
→ socket based NW app

→ multiple users simult

Working with Blocking call

Getting Started

Server



Server Socket

use builtin socket module.

socket params

*AF_INET → IP proto

*SOCK_STREAM → TCP

* SO_REUSEADDR → reuse port after app stop

* Buffering at OS level is possible

3.2.1

each time we call socket connect recvfrom method we get a new client.

but ~~see~~ conn.recv() is a blocking call

3.3 Possibility of non blocking sockets

socket.setblocking(False)

when conn.recv() is called if no data it throws exceptions

But this approach causes a high looping so sleep is needed for CPU load lessing

These notification systems of os are the core of how asyncio achieves concurrency. Understanding how this works gives us a view of how the underlying machinery of asyncio works.

SELECTORS → Python package

↳ helps to know OS events

... → default selector class

chooses best implementation of event notification system

3.5.3

```
selector.register(server_socket, selectors.EVENT_READ)
```

[Select] will block until an event has happen, the call will return with a list of sockets that are ready 4 processing, also supports a timeout

* create socket
* * make socket non blocked
  * register selector on socket
* listen for event on selector
  ↳ blocking with timeout

```
for evn in events
    esen_socket = evn.fileobj # esen_file obj
if event_sock == server_sock
    selector.register(connected)
else data = esen_sock.recv
```

this example was how a asyncio works

There are three main coroutines we'll want to work with: sock_accept, sock_recv and sock_sendall. These are analogous to the socket methods that we used earlier, except that they take in a socket as an argument and return coroutines that we can await until we have data to act on.

sock_accept. This coroutine is analogous to the socket.accept method that we saw in our first implementation. This method will return a tuple (a data structure that stores an ordered sequence of values) of a socket connection and a client address. We pass it in the socket we're interested in, and we can then await the coroutine it returns. Once that coroutine completes, we'll have our connection and address. This socket must be non-blocking and should already be bound to a port:

* next is to try as coroutine + tasks

process one connection at a time using socket.accept
↳ other connection are stored in queue

we put the while inside a coroutine to wait for connection

* to write + read data in each connection we need a task ~~for each~~ connection

```
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_sock
        connection.setblocking(False)
        print(f"Got a connection from {address}")
        asyncio.create_task(echo(connection, loop))
```

What happens when exception occurs inside a task

exception Task exception was never retrieved
↳ Task is considered done

The If we await a task the exception will get thrown where we perform the await

If we keep a reference of the task obj, the exception is not thrown since the traceback of failed occurs when the garbage collector is triggered

Use of try catch within task is a best practice

→ async.run when we cancel a task CancelledError exception is raised
↳ this could be due to exception

Shutting Down Gracefully

Use of Signals

listening for signals SIGTERM + SIGINT

async event loop listens 4 event with its add_signal_handler method

This function takes in a signal we want to listen for and a function that we'll call when our application receives that signal.

```
def cancel_tasks():
    print('Got a SIGINT!')
    tasks: Set[asyncio.Task] = asyncio.all_tasks()
    print(f'Cancelling {len(tasks)} task(s).')
    [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)
```

There is a asyncio.all_tasks

Look at the syntax : tasks : Set[asyncio.Task] = asyncio.all_tasks()

3.6.2 Waiting 4 pending task to finish

the cancel_tasks() method is a normal function means we can't await inside it .

So we can pass a task to await all Tasks, to the sig handler function

* if await _all_tasks raises exception we won't catch it . instead run main as a coroutine in a task & create event loop instead of asyncio.run

WAIT_FOR → Tasks with timeout

Concurent web request

lib AIOHttp
↳ this gives coroutines → which we can then start or await

Request lib has blocking calls

AIOHTTP Server & Client

Async Contexct Managers [With STMT]

Without With

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

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

This is cleaner "with" statement, But this is working in sync program

Async with

↳ they implement __aenter__ & __aexit__

__aenter__ → Acquire resource

__aexit__ → close resource

We can override __aenter__ & __aexit__ with our own implementation

```
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):                                          ❶
        print('Entering context manager, waiting for connection')
        loop = asyncio.get_event_loop()
        connection, address = await loop.sock_accept(self._server_socket)
        self._connection = connection
        print('Accepted a connection')
        return self._connection

    async def __aexit__(self,
                        exc_type: Optional[Type[BaseException]],
                        exc_val: Optional[BaseException],
                        exc_tb: Optional[TracebackType]):                ❷
        print('Exiting context manager')
        self._connection.close()
        print('Closed connection')


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)
    server_socket.bind(server_address)
    server_socket.listen()

    async with ConnectedSocket(server_socket) as connection:             ❸
        data = await loop.sock_recv(connection, 1024)
        print(data)                                                      ❹


asyncio.run(main())
```

From <https://learning.oreilly.com/library/view/python-concurrency-with/9781617298660/OEBPS/Text/04.htm#sigil_toc_id_66>

4.2.1 Making web request with aiohttp

* Sessions Similar to a browser tab
* Connection pooling session has connections [session has a
* Session obj → has Get, post, put method

```python
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'Status for {url} was {status}')


asyncio.run(main())
```

From <https://learning.oreilly.com/library/view/python-concurrency-with/9781617298660/OEBPS/Text/04.htm#sigil_toc_id_87>

Session has a default of 100 connections → upper of concurrent requests → change using TCPConnector obj

AIOHTTP windows event loop may need some work arro

4.2.2 Timeout with aiohttp

We can do this similar to how we did for async.wait_for.

AIOHTTP — has a cleaner way to do it. using Client Timeout data

Client Timeout
↳ Timeout for sessions
Timeout for connections

```python
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')
```

→ connection Timeout

→ session Timeout

asyncio.run(main())

4.3 Running tasks concurrently using list comprehension

```
tasks = [asyncio.create_task(delay(seconds)) for seconds in [2,3,5]
[await task for task in tasks]
```

Disadvantages

* code lines
* if 1 coroutine finish first it still has to wait
* exception handling is not easy

4.4 Runing request with Gather

* gives an opt param [return_exceptions]

→ False → gather will throw when we await, I 1 coroutine fails everything is cancelled

→ True → returns a list of exceptions

```
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)]
```

Drawbacks

→ not easy to cancel tasks

Drawbacks

* not easy to cancel tasks
* we must wait 4 all coroutine before ca

can process any result

* Processing request as they complete [4.5]

as_completed → returns an iterator of futures

When we await the futures, the first coroutine which got completed will be returned first

```
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)
```

45.1. Timeouts with as_completed.

```
        for done_task in asyncio.as_completed(fetchers, timeout=2):
            try:
                result = await done_task
                print(result)
            except asyncio.TimeoutError:
                print('We got a timeout error!')
```

This a timeout for each coroutine

From <https://learning.oreilly.com/library/view/python-concurrency-with/9781617298660/OEBPS/Text/04.html#sec_ch04_lvl1_74>.

Drawback:

which tasks are still running, which are finish

4.6. Fine grained control with wait.

```
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'Done task count: {len(done)}')
        print(f'Pending task count: {len(pending)}')

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


asyncio.run(main())
```

see the use of wait

see the return "done & pending"

From <https://learning.oreilly.com/library/view/python-concurrency-with/9781617298660/OEBPS/Text/04.htm#sigil_toc_id_75>

Done includes → sucess & exception

Pending not yet over

ALL_Completed flag [Pending is 0]

Handling Exception in the done list

```
done, pending = await asyncio.wait(fetchers)

    print(f'Done task count: {len(done)}')
    print(f'Pending task count: {len(pending)}')

    for done_task in done:
        # result = await done_task will throw an exception
        if done_task.exception() is None:
            print(done_task.result())
        else:
            logging.error("Request got an exception",
                          exc_info=done_task.exception())
```

see we await a done task this is to raise exception

4.6.2 Watching 4 Exceptions

First_EXEMPTION flag

↳ If no exception similar to ALL_COMPLETED

If exception the wait will immediately return at first Exception

4.6.3 Processing request as they complete

FIRST_COMPLETED

↳ will make wait coroutine return as soon as atleast One result

```
import asyncio
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'Done task count: {len(done)}')
            print(f'Pending task count: {len(pending)}')

            for done_task in done:
                print(await done_task)

asyncio.run(main())
```

Notice we are waiting for pending tasks

4.6.4 handling timeout

we can give timeout to asyncio.wait() but it will not cancel task as in the case with gather or as-completed

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

From <https://learning.oreilly.com/library/view/python-concurrency-with/9781617298660/OEBPS/Text/04.htm#sigil_toc_id_79>

4.6.5 Why wrap in task

If we call wait with co routine they are automatically wrapped in tasks, & the return in done & pending are tasks

* But we can't do any comparison with the task obj, as we will have the obj we created task.