About Python Async IO

Backend.ai 이태우

목차

Part 1,

AsyncIO와 Thread

Part 1,

Python내의 Concurrency 구현 방법

Multiprocessing, Thread, AsyncIO

1 **Multiprocessing : 다중코어 병렬처리**

2 **Thread : GIL 상태의 단일코어 다중 Thread**

GIL : Global Interpreter Lock

병렬 처리를 위해선 **Thread** 혹은 **Process**의
추가적인 생성을 위한 추가 소요시간 발생

3 **AsyncIO : 단일 Thread Concurrency**

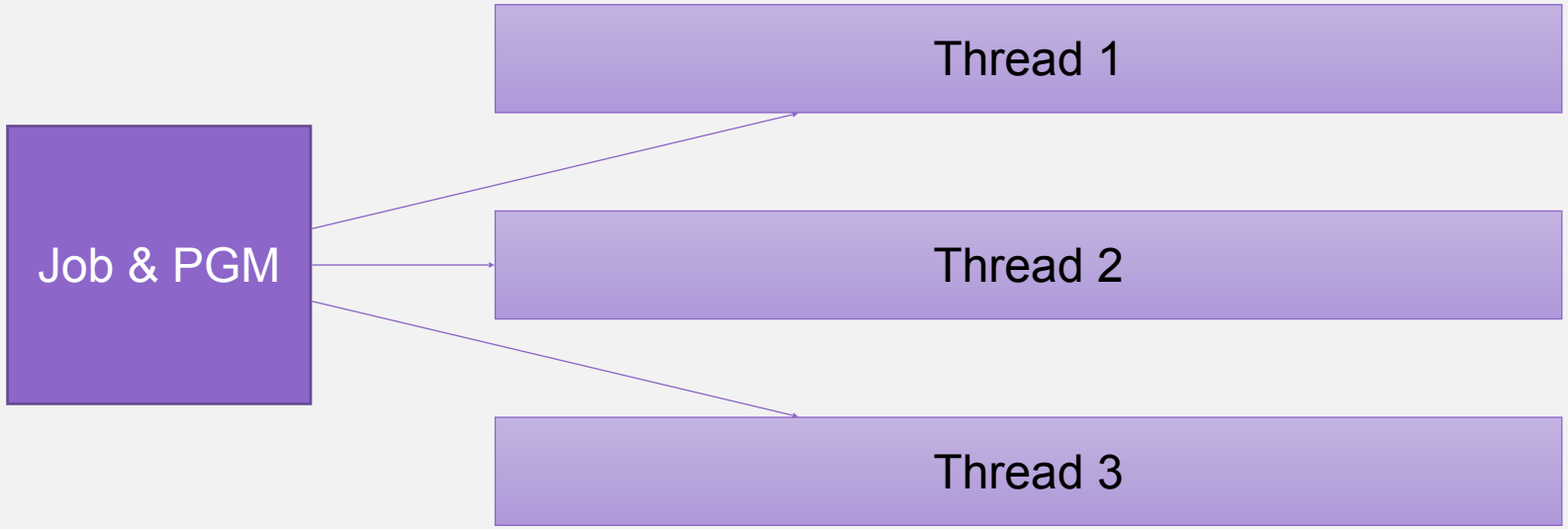
Part 1,

Thread와 AsyncIO의 차이점

Difference between Thread and AsyncIO

Thread

OS 내의 Thread를 추가로 생성하여 동시성 Program을 구현
→ 동시에 실행되는 Program 개수만큼 Thread의 추가가 요구됨
EX. Thread의 Pool을 1(단일 Thread)로 지정할 경우 일반적인 프로그램이랑 똑같음



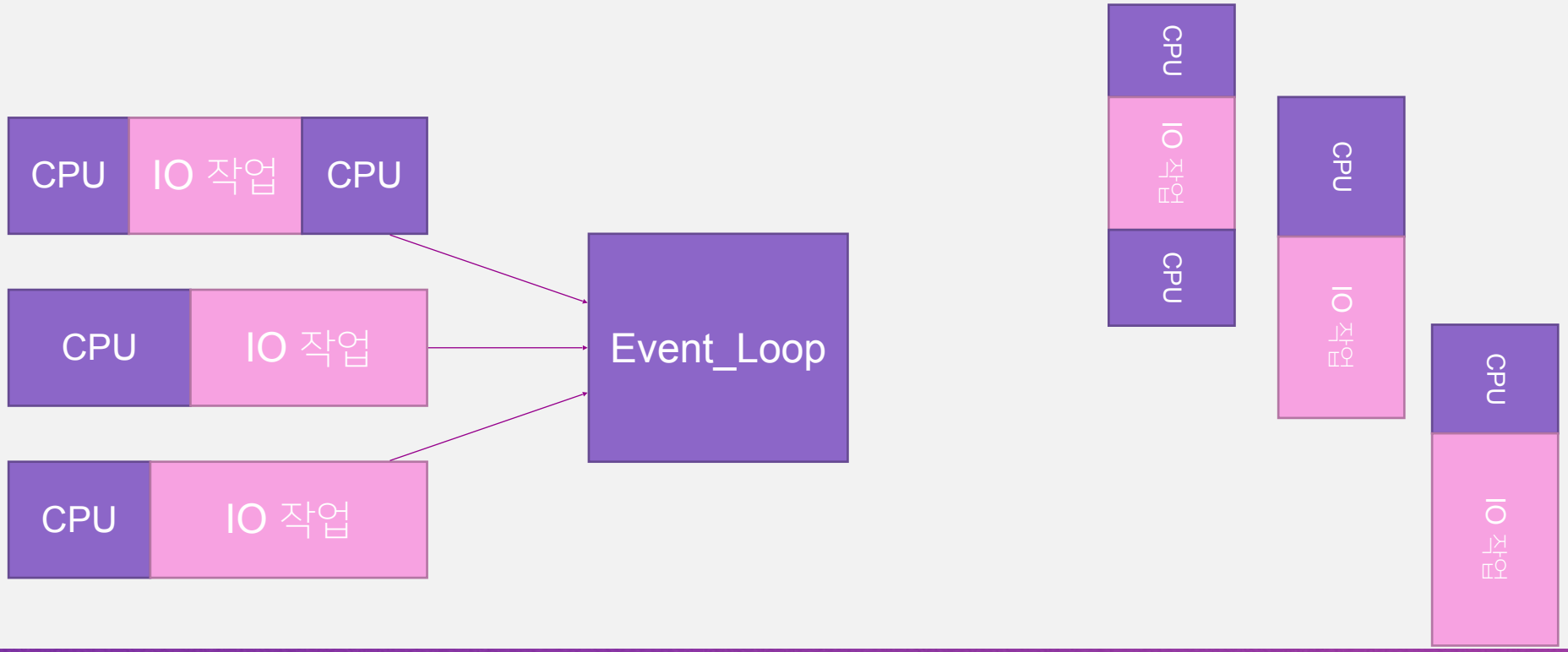
Part 1,

Thread와 AsyncIO의 차이점

Difference between Thread and AsyncIO

AsycnIO

단일 Thread 내에서 Event_Loop를 통해서 여러 Program의 동시성 구현
→ 모든 작업들은 Awaitable 해야됨
→ IO 작업은 CPU연산이 동반되지 않는 작업을 의미
보조기억장치 —> 주기억장치로 Data 이동 or Requests를 보내고 Response를 기다리기



Part 2,

AsyncIO의 사용 패턴

Part 2,

Python Asyncio의 진행 패턴

Lorem Ipsum is simply dummy text of the printing and typesetting industry.

AsyncIO High Level API를 사용한 Async동작

High Level 직관성을 위해서
3.10부터 AsyncIO에서 사라짐

코루틴 정의	Event Loop 획득	코루틴 배치	코루틴 동작
def → asycn def	get_running_loop	create_task	Asyncio.run
	get_event_loop	gather	

Part 1,

코루틴 정의하기

Coroutine function/class context

Coroutine Method

def → async def

Await awitable obj로 이벤트루프 반환

Await는 async def 내에서만 사용 가능

Coroutine Class

Class 내부에 def __await__ 정의

생성자 → __await__ 순으로 실행

코루틴매쏘드.__await__() 형태로 return

```python
import asyncio

async def count():
    print("One")
    await asyncio.sleep(1)
    print("Two")

async def main():
    await asyncio.gather(count(), count(), count())

if __name__ == "__main__":
    import time
    s = time.perf_counter()
    asyncio.run(main())
    elapsed = time.perf_counter() - s
    print(f"{__file__} executed in {elapsed:0.2f} seconds.")
```

```python
class Hello:
    def __init__(self):
        print("init")

    def __await__(self):
        async def closure():
            asyncio.sleep(5)
            return self
        return closure().__await__()

    async def method(self):
        print("method")

async def main():
    h = await Hello()
    await h.method()
```

Part 2,

코루틴 배치

Coroutine function/class context

Create_task

Task Object를 생성 후 변수에 할당

Await를 사용해서 Event Loop로 전송

```python
async def main():
    task1 = asyncio.create_task(
        say_after(1, 'hello'))

    task2 = asyncio.create_task(
        say_after(2, 'world'))

    await task1
    await task2

    #await say_after(1, 'hello')
    #await say_after(2, 'world')
    #랑은 다릅니다.
```

Gather

Task의 생성 없이 Awaitable Object를 모아서 Event Loop에 전송

```python
async def main():
    await asynio.gather(say_after(1, 'hello'),
                        say_after(2, 'world'),
                        )
```

Part 3,

코루틴 동작

Start Coroutine

AsyncIO.run

일반적으로 코루틴 생성 → IO 작업 배치 → IO작업이 배치된 코루틴은 실행 의 순서로 Code가 쓰여짐

```python
import asyncio

async def count():
    print("One")
    await asyncio.sleep(1)
    print("Two")

async def main():
    await asyncio.gather(count(), count(), count())

if __name__ == "__main__":
    import time
    s = time.perf_counter()
    asyncio.run(main())
    elapsed = time.perf_counter() - s
    print(f"{__file__} executed in {elapsed:0.2f} seconds.")
```

Part 4,

Async with / Async for

Async Context Mange

Async with

Class 내부에
async def __aenter__(self)
async def __aexit__(self) 의 정의 필요

Async for

Class 내부에
def __aiter__(self)
async def __anext__(self)의 정의 필요

```
class Hello:
    def __init__(self):
        print("init")

    async def __aenter__(self):
        print("enter")
        await ayncio.sleep(3600)
        return self

    async def __aexit__(self, *args):
        print("exit")
                    Hello Class의 생성자 이후
                    Context에 들어가기 전 Await이 가능
async def main():
    async with Hello() as h:
        print("context")

if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.run_until_complete(main())
```

```
class Reader:
    #an awaitable Coroutine
    #that generate some output
    async def readline(self):
        ...
                해당 객체가 iterator가 되게만듦
    def __aiter__(self):
        return self

    async def __anext__(self):
        val = await self.readline()
        if val == b'':
            raise StopAsyncIteration
        return val
```

Part 3

Backend.ai Async 코드 패턴

Part 1,

Backend.ai-client-py의 image.py rescan

…

```python
@image.command()
@click.option('-r', '--registry', type=str, default=None,
              help='The name (usually hostname or "lablup") '
                   'of the Docker registry configured.')
def rescan(registry: str) -> None:

    async def rescan_images_impl(registry: str) -> None:
        async with AsyncSession() as session:  2
            try:
                result = await session.Image.rescan_images(registry)
            except Exception as e:
                print_error(e)
                sys.exit(1)
            if not result['ok']:
                print_fail(f"Failed to begin registry scanning: {result['msg']}")
                sys.exit(1)
            print_done("Started updating the image metadata from the configured registries.")
            task_id = result['task_id']
            bgtask = session.BackgroundTask(task_id)
            ...
            ...

            finally:
                completion_msg_func()

    asyncio_run(rescan_images_impl(registry))  1
```

1. 일반 함수(rescan) 내부에서 await를 사용하기 위해서 내부에서 코루틴 정의 후 Event Loop를 생성하여 코루틴을 실행

```python
def _asyncio_run(coro, *, debug=False):
    loop = asyncio.new_event_loop()
    asyncio.set_event_loop(loop)
    loop.set_debug(debug)
    try:
        return loop.run_until_complete(coro)
    finally:
        try:
            _cancel_all_tasks(loop)
            if hasattr(loop, 'shutdown_asyncgens'):  # Python 3.6+
                loop.run_until_complete(loop.shutdown_asyncgens())
        finally:
            loop.stop()
            loop.close()
            asyncio.set_event_loop(None)


if hasattr(asyncio, 'run'):  # Python 3.7+
    asyncio_run = asyncio.run
else:
    asyncio_run = _asyncio_run
```

2. AsyncSession의 Context 생성을 위한 Async with 구문

Part 1,

Backend.ai-client-py의 session.py AsyncSession

…

```python
class AsyncSession(BaseSession):

    def __init__(
        self, *,
        config: APIConfig = None,
        proxy_mode: bool = False,
    ) -> None:
        ...
        ...

    async def _aopen(self) -> None:
        self._context_token = api_session.set(self)
        if not self._proxy_mode:
            self.api_version = await _negotiate_api_version(self.aiohttp_session, self.config)

    def open(self) -> Awaitable[None]:
        return self._aopen()

    async def _aclose(self) -> None:
        if self._closed:
            return
        self._closed = True
        await _close_aiohttp_session(self.aiohttp_session)
        api_session.reset(self._context_token)

    def close(self) -> Awaitable[None]:
        return self._aclose()

    async def __aenter__(self) -> AsyncSession:
        assert not self.closed, 'Cannot reuse closed session'
        await self.open()
        if self.config.announcement_handler:
            try:
                payload = await self.Manager.get_announcement()
                if payload['enabled']:
                    self.config.announcement_handler(payload['message'])
            except (BackendClientError, BackendAPIError):
                # The server may be an old one without annoucement API.
                pass
        return self

    async def __aexit__(self, *exc_info) -> Literal[False]:
        await self.close()
        return False  # false up the inner exception
```

3. Aysnc Context를 구현하기 위해서 AsyncSession Class 내부에 Async def __aenter__(self), async def __aexit__(self)를 구현

이때 특이한 패턴으로 Async def로 코루틴을 정의하고, 일반 함수의 반환을 코루틴으로 반환하여 Awaitable한 object를 return하게함

Part 1,

부록. Backend.ai의 특별한 Async Class

…

```python
class aobject(object):
    '''
    An "asynchronous" object which guarantees to invoke both ``def __init__(self, ...)`` and
    ``async def __ainit(self)__`` to ensure asynchronous initialization of the object.

    You can create an instance of subclasses of aboject in the following way:

    .. code-block:: python

       o = await SomeAObj.new(...)
    '''

    @classmethod
    async def new(cls: Type[T_aobj], *args, **kwargs) -> T_aobj:
        '''
        We can do ``await SomeAObject(...)``, but this makes mypy
        to complain about its return type with ``await`` statement.
        This is a copy of ``__new__()`` to workaround it.
        '''
        instance = super().__new__(cls)
        instance.__init__(*args, **kwargs)
        await instance.__ainit__()
        return instance

    def __init__(self, *args, **kwargs) -> None:
        pass

    async def __ainit__(self) -> None:
        '''
        Automatically called when creating the instance using
        ``await SubclassOfAObject(...)``
        where the arguments are passed to ``__init__()`` as in
        the vanilla Python classes.
        '''
        pass
```

4. Python의 __await__을 정의하더라도, Context Manger를 사용할 경우 생성자 실행 -> __aenter__ 실행만 되고, __await__부분의 실행이 안됨.
해당 부분 방지하기 위한 것으로 추정 및 Mypy대응 및 Awaitable Object를 만들기 위한 aobject Class를 commo에 만들고 상속받는 방법으로 ainit을 override함

감사합니다