使用asyncio

官网对asyncio的描述

Asynchronous I/O, event loop, coroutines and tasks

asyncio的生态问题

1. 迁移成本太高
2. 使用后的效果不突出
3. asyncio改变了编程习惯
4. 没有大公司出来背书

```python
import asyncio


async def coroutine():
    print('in coroutine')
    return 'result'


event_loop = asyncio.get_event_loop()
try:
    print('starting coroutine')
    coro = coroutine()
    print('entering event loop')
    result = event_loop.run_until_complete(coro)
    print(f'it returned: {result}')
finally:
    print('closing event loop')
    event_loop.close()
```

```
❯ python asyncio_simple.py
starting coroutine
entering event loop
in coroutine
it returned: result
closing event loop
```

```python
import asyncio


async def main():
    print('waiting for chain1')
    result1 = await chain1()
    print('waiting for chain2')
    result2 = await chain2(result1)
    return (result1, result2)


async def chain1():
    print('in chain1')
    return 'result1'


async def chain2(arg):
    print('in chain1')
    return 'Derived from {}'.format(arg)


event_loop = asyncio.get_event_loop()
try:
    return_value = event_loop.run_until_complete(main())
    print(f'return value: {return_value}')
finally:
    event_loop.close()
```

```
❯ python asyncio_chain.py
waiting for chain1
in chain1
waiting for chain2
in chain2
return value: ('result1',
    'Derived from result1')
```

旧式写法

```
import asyncio


@asyncio.coroutine
def main():
    print('waiting for chain1')
    result1 = yield from chain1()
    print('waiting for chain2')
    result2 = yield from chain2(result1)
    return (result1, result2)


@asyncio.coroutine
def chain1():
    print('in chain1')
    return 'result1'


@asyncio.coroutine
def chain2(arg):
    print('in chain2')
    return 'Derived from {}'.format(arg)


event_loop = asyncio.get_event_loop()
try:
    return_value = event_loop.run_until_complete(main())
    print(f'return value: {return_value}')
finally:
    event_loop.close()
```

async with

```python
# 需要先安装aiohttp:  pip install aiohttp
import asyncio
import aiohttp


async def fetch_page(url):
    async with aiohttp.ClientSession() as session:
        async with session.get(url) as response:
            return await response.json()


loop = asyncio.get_event_loop()
result = loop.run_until_complete(
    fetch_page('http://httpbin.org/get?a=2'))
print(f"Args: {result.get('args')}")
loop.close()
```

```
❯ python asyncio_fetch.py
Args: {'a': '2'}
```

async for

```
import asyncio


async def g1():
    yield 1
    yield 2


async def g2():
    async for v in g1():
        print(v)
    return [v * 2 async for v in g1()]


loop = asyncio.get_event_loop()
try:
    loop.run_until_complete(g2())
finally:
    loop.close()
```

```
❯ python asyncio_for.py
1
2
Result is [2, 4]
```

Future & Task

```python
import asyncio


async def func1():
    await asyncio.sleep(1)
    await func(1)


async def func2():
    await func(2)


async def func(num):
    print(num * 2)


loop = asyncio.get_event_loop()
tasks = asyncio.gather(
    asyncio.ensure_future(func1()),
    asyncio.ensure_future(func2())
)
loop.run_until_complete(tasks)

tasks = [
    asyncio.ensure_future(func1()),
    asyncio.ensure_future(func2())
    ]
loop.run_until_complete(asyncio.wait(tasks))
loop.close()
```

```
❯ python asyncio_future.py
4
2
4
2
```

同步机制

1. Semaphore(信号量)

```python
import aiohttp
import asyncio

NUMBERS = range(6)
URL = 'http://httpbin.org/get?a={}'
sema = asyncio.Semaphore(3)


async def fetch_async(a):
    async with aiohttp.request('GET', URL.format(a)) as r:
        data = await r.json()
    return data['args']['a']


async def print_result(a):
    with (await sema):
        r = await fetch_async(a)
        print('fetch({}) = {}'.format(a, r))


loop = asyncio.get_event_loop()
f = asyncio.wait([print_result(num) for num in NUMBERS])
loop.run_until_complete(f)
loop.close()
```

```
❯ python asyncio_semaphore.py
fetch(4) = 4
fetch(3) = 3
fetch(1) = 1
fetch(2) = 2
fetch(0) = 0
fetch(5) = 5
```

2. Lock(锁)

```
import asyncio
import functools


def unlock(lock):
    print('callback releasing lock')
    lock.release()


async def test(locker, lock):
    print('{} waiting for the lock'.format(locker))
    with await lock:
        print('{} acquired lock'.format(locker))
    print('{} released lock'.format(locker))


async def main(loop):
    lock = asyncio.Lock()
    await lock.acquire()
    loop.call_later(0.1, functools.partial(unlock, lock))
    await asyncio.wait([test('l1', lock), test('l2', lock)])

loop = asyncio.get_event_loop()
loop.run_until_complete(main(loop))
loop.close()
```

```
❯ python asyncio_lock.py
l2 waiting for the lock
l1 waiting for the lock
callback releasing lock
l2 acquired lock
l2 released lock
l1 acquired lock
l1 released lock
```

3. Condition(条件)

```python
import asyncio
import functools

async def consumer(cond, name, second):
    await asyncio.sleep(second)
    with await cond:
        await cond.wait()
        print('{}: Resource is available to consumer'.format(name))

async def producer(cond):
    await asyncio.sleep(2)
    for n in range(1, 3):
        with await cond:
            print('notifying consumer {}'.format(n))
            cond.notify(n=n)
        await asyncio.sleep(0.1)

async def producer2(cond):
    await asyncio.sleep(2)
    with await cond:
        print('Making resource available')
        cond.notify_all()

async def main(loop):
    condition = asyncio.Condition()
    task = loop.create_task(producer(condition))
    consumers = [consumer(condition, name, index)
                 for index, name in enumerate(('c1', 'c2'))]
    await asyncio.wait(consumers)
    task.cancel()
    task = loop.create_task(producer2(condition))
    consumers = [consumer(condition, name, index)
                 for index, name in enumerate(('c1', 'c2'))]
    await asyncio.wait(consumers)
    task.cancel()

loop = asyncio.get_event_loop()
loop.run_until_complete(main(loop))
loop.close()
```

```
❯ python asyncio_condition.py
notifying consumer 1
c1: Resource is available to consumer
notifying consumer 2
c2: Resource is available to consumer
Making resource available
c1: Resource is available to consumer
c2: Resource is available to consumer
```

4. Event(事件)

```python
import asyncio
import functools


def set_event(event):
    print('setting event in callback')
    event.set()


async def test(name, event):
    print('{} waiting for event'.format(name))
    await event.wait()
    print('{} triggered'.format(name))


async def main(loop):
    event = asyncio.Event()
    print('event start state: {}'.format(event.is_set()))
    loop.call_later(
        0.1, functools.partial(set_event, event)
    )
    await asyncio.wait([test('e1', event), test('e2', event)])
    print('event end state: {}'.format(event.is_set()))

loop = asyncio.get_event_loop()
loop.run_until_complete(main(loop))
loop.close()
```

```
❯ python asyncio_event.py
event start state: False
e2 waiting for event
e1 waiting for event
setting event in callback
e2 triggered
e1 triggered
event end state: True
```

5. 队列(Queue)

```
import asyncio
import random
import aiohttp

NUMBERS = random.sample(range(100), 7)
URL = 'http://httpbin.org/get?a={}'
sema = asyncio.Semaphore(3)


async def fetch_async(a):
    async with aiohttp.request('GET', URL.format(a)) as r:
        data = await r.json()
    return data['args']['a']


async def collect_result(a):
    with (await sema):
        return await fetch_async(a)


async def produce(queue):
    for num in NUMBERS:
        print('producing {}'.format(num))
        item = (num, num)
        await queue.put(item)


async def consume(queue):
    while 1:
        item = await queue.get()
        num = item[0]
        rs = await collect_result(num)
        print('consuming {}...'.format(rs))
        queue.task_done()
```

```
async def run():
    queue = asyncio.PriorityQueue()
    consumer = asyncio.ensure_future(consume(queue))
    await produce(queue)
    await queue.join()
    consumer.cancel()


loop = asyncio.get_event_loop()
loop.run_until_complete(run())
loop.close()
```

```
❯ asyncio_queue.py
producing 37
producing 47
producing 26
producing 49
producing 76
producing 45
producing 8
consuming 8...
consuming 26...
consuming 37...
consuming 45...
consuming 47...
consuming 49...
consuming 76...
```

延伸阅读

1.https://docs.python.org/3/library/asyncio.html

2.https://pymotw.com/3/asyncio/index.html

3.https://djangostars.com/blog/asynchronous-programming-in-python-asyncio/

4.http://www.dongwm.com/archives/%E4%BD%BF%E7%94%A8Python%E8%BF%9B%E8%A1%8C%E5%B9%B6%E5%8F%91%E7%BC%96%E7%A8%8B-%E6%88%91%E4%B8%BA%E4%BB%80%E4%B9%88%E4%B8%8D%E5%96%9C%E6%AC%A2Gevent/

5.https://www.python.org/dev/peps/pep-0492/

6.https://www.python.org/dev/peps/pep-3156/

7.http://www.dongwm.com/archives/%E4%BD%BF%E7%94%A8Python%E8%BF%9B%E8%A1%8C%E5%B9%B6%E5%8F%91%E7%BC%96%E7%A8%8B-asyncio%E7%AF%87/

8.https://www.python.org/dev/peps/pep-0525/

9.http://asyncio.readthedocs.io/en/latest/producer_consumer.html

10.http://aosabook.org/en/500L/a-web-crawler-with-asyncio-coroutines.html