2017-05-25 Thu 11:18, by 田宇伟

生成器、协程和asyncio（下）

目录

1 关闭 loop

上面的例子都没有调用 `loop.close` ，loop只要不关闭，就还可以再运行。

```
loop.run_until_complete(coro(loop, 3))
loop.run_until_complete(coro(loop, 2))

loop.close()
```

建议调用 `loop.close` ，以彻底清理 `loop` 对象防止误用。

2 coroutine 和 future

```
result = await future
result = await coroutine
```

coroutine 是生成器函数，它既可以从外部接受参数，也可以产生结果。使用 coroutine 的好处是可以暂停一个函数，然后稍后恢复执行。在停下的这段时间内，可以切换到其他任务继续执行。

future 更像是一个占位符，其值会在将来被计算出来。例如在等待网络 IO 函数完成时，函数会返回一个容器，future 会在完成时填充该容器。填充完毕后，可以用回调函数来获取实际结果。

`Task` 对象是 `Future` 的子类，它将 coroutine 和 future 联系在一起，将 coroutine 封装成一个 `Future` 对象。

两种任务启动方法：

```
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))
```

`ensure_future` 可以将 coroutine 封装成 `Task` 类型 。 `asyncio.gather` 将 Future 和 coroutine 封装成一个 `Future` 对象 。 `asyncio.wait` 则本身就是 coroutine 。

关于 `asyncion.gather()` 和 `asyncio.wait()` 的比较，可具体参考 StackOverFlow 。

`run_until_complete` 既可以接收 `Future` 对象，也可以是 coroutine 对象。

```
BaseEventLoop.run_until_complete(future)

Run until the Future is done.

If the argument is a coroutine object, it is wrapped by ensure_future().

Return the Future's result, or raise its exception.
```

3 Task 任务退出

根据官方文档， Task 对象只有在以下几种情况，会认为是退出：

```
a result / exception are available, or that the future was cancelled
```

Task 对象的 cancel 和其父类 Future 略有不同。当调用 Task.cancel() 后，对应 coroutine 会在事件循环的下一轮中抛出 CancelledError 异常。使用 Future.cancelled() 并不能立即返回 True（用来表示任务结束），只有在上述异常被处理任务结束后才算是 cancelled。

结束任务：

```python
for task in asyncio.Task.all_tasks():
    task.cancel()
```

这种方法将所有任务找出并 cancel。

但 CTRL-C 也会将事件循环停止，所以有必要重启事件循环，

```python
try:
    loop.run_until_complete(tasks)
except KeyboardInterrupt as e:
    for task in asyncio.Task.all_tasks():
        task.cancel()
        loop.run_forever()  # restart loop
finally:
    loop.close()
```

在每个 Task 中捕获异常是必要的，如果不确定，可以使用 asyncio.gather(..., return_exceptions=True) 将异常转换为正常的结果返回。

4 Timer

```python
import asyncio


async def timer(wt, cb):
    ft = asyncio.ensure_future(asyncio.sleep(wt))
    ft.add_done_callback(cb)
    print("Waiting {}s ...".format(wt))
    await ft

t = timer(30, lambda ft: print('>>> Done'))
loop = asyncio.get_event_loop()
loop.run_until_complete(t)
```

```
Waiting 30s ...
>>> Done
```

5 async for / async with

async for 是异步迭代器语法。为支持异步迭代，异步对象需实现 __aiter__ 方法，异步迭代器需实现 __anext__ 方法，停止迭代需在 __anext__ 方法内抛出 StopAsyncIteration 异常。

```python
# async_for.py
import random
import asyncio
```

```
class AsyncIterable:
    def __init__(self):
        self.count = 0

    async def __aiter__(self):
        return self

    async def __anext__(self):
        if self.count >= 5:
            raise StopAsyncIteration
        data = await self.fetch_data()
        self.count += 1
        return data

    async def fetch_data(self):
        return random.choice(range(10))


async def main():
    async for data in AsyncIterable():
        print(data)


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

```
0
8
3
9
5
```

async with 是异步上下文管理器语法，需实现 __aenter__ 和 __aexit__ 方法。

```
# async_with.py
async def log(msg):
    print(msg)


class AsyncContextManager:
    async def __aenter__(self):
        await log('entering context')

    async def __aexit__(self, exc_type, exc, tb):
        await log('exiting context')


async def coro():
    async with AsyncContextManager():
        print('body')


c = coro()
try:
    c.send(None)
except StopIteration:
    print('finished')
```

```
entering context
body
exiting context
finished
```

6 示例

```
import aiohttp
import asyncio


async def get_status(url, id):
    async with aiohttp.ClientSession() as session:
        async with session.get(url) as r:
            print(r.status, id)


tasks = []
for i in range(10):
    ft = asyncio.ensure_future(get_status('https://api.github.com/events', id=i))
    tasks.append(ft)

loop = asyncio.get_event_loop()
```

```
loop.run_until_complete(asyncio.wait(tasks))
loop.close()
```

```
200 5
200 0
200 2
200 8
200 1
200 7
200 6
200 3
200 4
200 9
```

6.1 示例2

```
import asyncio
import itertools
import sys


async def spin(msg):
    write, flush = sys.stdout.write, sys.stdout.flush
    for char in itertools.cycle('|/-\\'):
        status = char + ' ' + msg
        write(status)
        flush()
        write('\x08' * len(status))
        try:
            await asyncio.sleep(.1)
        except asyncio.CancelledError:
            break
        write(' ' * len(status) + '\x08' * len(status))


async def slow_function():
    await asyncio.sleep(10)
    return 42


async def supervisor():
    spinner = asyncio.ensure_future(spin('thinking!'))
    print('spinner object:', spinner)
    result = await slow_function()
    spinner.cancel()
    return result


def main():
    loop = asyncio.get_event_loop()
    result = loop.run_until_complete(supervisor())
    loop.close()
    print('Answer:', result)


main()
```

6.2 示例3

```
import asyncio

import aiohttp
import tqdm


async def print_page(url):
    text = await get(url)
    return text


async def get(*args, **kwargs):
    response = await aiohttp.request('GET', *args, **kwargs)
    return response.status


async def wait_with_progress(coros):
    for f in tqdm.tqdm(asyncio.as_completed(coros), total=len(coros)):
        await asyncio.sleep(2)
        await f


loop = asyncio.get_event_loop()
tasks = []
for i in range(10):
    ft = asyncio.ensure_future(print_page('http://httpbin.org/get'))
```

```
    tasks.append(ft)
    loop.run_until_complete(wait_with_progress(tasks))
```

```
100%|██████████████████████████████████████████| 10/10 [00:20<00:00,  2.00s/it]
```

6.3 示例4

```
import aiohttp
import asyncio

NUMBERS = range(12)
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)
```

```
fetch(5) = 5
fetch(1) = 1
fetch(10) = 10
fetch(8) = 8
fetch(9) = 9
fetch(3) = 3
fetch(7) = 7
fetch(11) = 11
fetch(2) = 2
fetch(6) = 6
fetch(4) = 4
fetch(0) = 0
```