协程 & asyncio & 异步编程

B站课程：Python异步编程---协程 & asyncio & 异步

01 课程介绍*哔哩哔哩*bilibili

1.协程

协程不是计算机提供。程序员人为创造。

协程（Coroutine），也可以被称为微线程，是一种用户态的上下文切换技术。简而言之，其实就是通过一个线程实现代码相互切换执行，例如：

```
def func1():
    print(1)

def func2():
    print(2)

func1()
func2()
```

实现协程的几种方法

- greenlet，早期模块
- yield关键字
- asyncio装饰器（3.4）
- async、await关键字（3.5）【推荐】

1.1 greenlet实现协程

```
from greenlet import greenlet


def func1():
    print(1)
    gr2.switch()
    print(2)
    gr2.switch()


def func2():
    print(3)
    gr1.switch()
    print(4)


gr1 = greenlet(func1)
```

```
gr2 = greenlet(func2)
gr1.switch()
'''
1
3
2
4
'''
```

1.2 yield 关键字

```
def func1():
    yield 1
    yield from func2()
    yield 2


def func2():
    yield 3
    yield 4


f1 = func1()
for item in f1:
    print(item)
'''
1
3
4
2
'''
```

1.3 asyncio

python3.4之后版本

```
import asyncio


@asyncio.coroutine
def func1():
    print(1)
    yield from asyncio.sleep(2)
    print(2)


@asyncio.coroutine
def func2():
    print(3)
    yield from asyncio.sleep(2)
```

```
    print(4)


tasks = [
    asyncio.ensure_future(func1()),
    asyncio.ensure_future(func2()),
]

loop = asyncio.get_event_loop()
loop.run_until_complete(asyncio.wait(tasks))
'''
1
3
2
4
'''
```

遇到IO阻塞可以自动切换

1.4 async、await 关键字

python3.5之后版本

```
import asyncio


async def func1():
    print(1)
    await asyncio.sleep(2)
    print(2)


async def func2():
    print(3)
    await asyncio.sleep(2)
    print(4)


tasks = [
    asyncio.ensure_future(func1()),
    asyncio.ensure_future(func2()),
]

loop = asyncio.get_event_loop()
loop.run_until_complete(asyncio.wait(tasks))
'''
1
3
2
4
'''
```

不能使用 create_task，因为 loop 还没创建

2. 协程的意义

在一个线程中如果遇到IO等待时间，线程不会等待，利用空闲时间的时候再去干点其他事情。

案例：下载图片（网络IO）

- 普通方式（同步）

```
import requests


def download_image(url):
    print('start')
    response = requests.get(url)
    file_name = url.rsplit('-')[-1]
    with open(file_name, mode='wb') as file_object:
        file_object.write(response.content)
    print('close')


if __name__ == '__main__':
    url_list = []
    for item in url_list:
        download_image(item)
```

- 协程方式（异步）

```
import aiohttp, asyncio


async def fetch(session, url):
    print('发送请求', url)
    async with session.get(url, verify_ssl=False) as response:
        content = await response.content.read()
        file_name = url.rsplit('-')[-1]
        with open(file_name, mode='wb') as file_object:
            file_object.write(content)


async def main():
    async with aiohttp.ClientSession() as session:
        url_list = []
        tasks = [asyncio.create_task(fetch(session, url)) for url in
url_list]
        await asyncio.wait(tasks)


if __name__ == '__main__':
    asyncio.run(main())
```

3.异步编程

3.1 事件循环

理解成一个死循环，去检测并测试执行某些代码

```
# 伪代码

任务列表=[任务1，任务2，任务3]

while True:
    可执行的任务列表，已完成的任务列表=去任务列表中检查所有的任务，将"可执行"和"已完成"的任务返回

    for 就绪任务 in 已准备就绪的任务列表:
        执行已就绪的任务

    for 已完成的任务 in 已完成的任务列表:
        在任务列表中移除已完成的任务

    如果任务列表中的任务都已完成，终止循环
```

```
import asyncio

#去生成或获取一个事件循环
loop = asyncio.get_event_loop()
#将任务放到任务列表中
loop.run_until_complete(任务)
```

3.2 快速上手

协程函数是指定义函数时使用async def 函数名的方式。

协程对象是指执行协程函数()得到的对象。

```
async def func():
    pass

result =  func()
```

注意：执行协程函数创建的对象，函数内部的代码不会执行

如果想要执行协程函数内部代码，必须要将协程对象交给事件循环来处理

```
import asyncio

async def func():
    print('Come on')

result =  func()

#python3.7以前
loop = asyncio.get_event_loop()
loop.run_until_complete(result) #等同于loop.run_until_complete(func())

#python3.7
asyncio.run(result)
```

3.3 await 关键字

await+可等待的对象（协程对象，Future对象和Task对象等类似IO等待的对象）

示例1：

```
import asyncio


async def func():
    print('come')
    response = await asyncio.sleep(2)
    print('结束', response)


asyncio.run(func())
```

示例2：

```
import asyncio


async def others():
    print('start')
    await asyncio.sleep(2)
    print('end')
    return '返回值'


async def func():
    print('执行函数内部代码')

    #遇到IO操作挂起当前协程，等IO操作完成后再继续往下执行。当前协程挂起时，事件循环可以去执行其他协程任务
    response = await others()

    print('IO请求结束，结果为：', response)
```

```
asyncio.run(func())
```

示例3：

```
import asyncio


async def others():
    print('start')
    await asyncio.sleep(2)
    print('end')
    return '返回值'


async def func():
    print('执行函数内部代码')

    response1 = await others()
    print('IO请求结束，结果为：', response1)

    #一个协程函数中可以有多个await，但只能依次执行
    response2 = await others()
    print('IO请求结束，结果为：', response2)


asyncio.run(func())
```

await就是等待对象的值得到结果后再继续往下走

3.4 Task对象

task是被用来并发地排定协程时间

当协程被通过诸如函数asyncio.create_task()来包装成一个Task，协程自动会被排定运行时间

本质上Task是帮住我们在事件循环中添加多个任务

包装协程成为Task对象的还有loop.create_task()或者ensure_future()函数。不建议手动实例化task对象

python3.7以前使用ensure_future

示例1：

```
import asyncio


async def func():
    print(1)
    await asyncio.sleep(2)
    print(2)
    return '返回值'
```

```python
async def main():
    print('执行函数内部代码')

    #create以后立即加入事件循环中
    #被调用时运行，不在created位置卡住，而在await位置卡住
    task1 = asyncio.create_task(func())
    task2 = asyncio.create_task(func())

    print('main 结束')
    #结果会出现再main结束这段文字之后
    #线程只在await位置卡住并等待切换
    #ret1和ret2的生成时间是一起的，因为task1和task2是并行
    ret1 = await task1

    ret2 = await task2

    print(ret1, ret2)


asyncio.run(main())
```

示例2：

```python
import asyncio


async def func():
    print(1)
    await asyncio.sleep(2)
    print(2)
    return '返回值'


async def main():
    print('执行函数内部代码')

    #将任务添加到任务列表，name参数指定任务名称
    tasks = [
        asyncio.create_task(func(), name='n1'),
        asyncio.create_task(func(), name='n2'),
    ]

    print('main 结束')

    #等待任务列表完成，返回元组
    #done表示所有完成任务的集合，pending表示尚未完成的
    #timeout参数表示等待时间
    #main函数包含多个task任务时，main函数内部使用await asyncio.wait(tasks)
    done, pending = await asyncio.wait(tasks, timeout=None)


    print(done)
```

```
asyncio.run(main())
```

示例3：

```
import asyncio


async def func():
    print(1)
    await asyncio.sleep(2)
    print(2)
    return '返回值'


tasks_list = [
    func(),
    func(),
]

asyncio.run(asyncio.wait(tasks_list))
```

tasks_list当出现在协程函数外时，不能使用create_task，因为这个时候循环还没有创建，会报错

只能以协程函数列表的形式直接传给asyncio.wait，会自动包装成一个任务列表

3.5 asyncio.Future对象

Future是一个特殊的底层可等待对象，用来代表一个同步运行的最终结果

Future是Task的基类，Task内部await结果的处理基于Future对象来的

示例1：

```
import asyncio


async def main():
    #获取当前事件循环
    loop = asyncio.get_running_loop()

    #创建一个任务（future对象），这个任务什么都不干
    fut = loop.create_future()

    #等待任务的最终结果（future对象），没有结果会一直等下去
    await fut


asyncio.run(main())
```

示例2：

```python
import asyncio


async def func():
    print(1)
    await asyncio.sleep(2)
    print(2)
    return '返回值'


async def main():
    print('执行函数内部代码')

    tasks_list = [
        asyncio.create_task(func(), name='n1'),
        asyncio.create_task(func(), name='n2'),
    ]

    print('main 结束')

    #等待任务列表完成，返回元组
    done, pending = await asyncio.wait(tasks_list, timeout=None)

    print(done)


asyncio.run(main())
```

示例3：

```python
import asyncio


async def set_after(fut):
    await asyncio.sleep(2)
    fut.set_result('666')


async def main():
    loop = asyncio.get_running_loop()

    #创建一个任务（future对象），没有绑定任何行为，这个任务永远不会结束
    fut = loop.create_future()

    #将fut这个future作为参数传给set_after，返回一个绑定行为的future
    #create_task实际传入的是一个协程
    await loop.create_task(set_after(fut))

    #等待fut获取结果，
    data = await fut
    print(data)
```

```
asyncio.run(main())
```

3.6 concurrent.futures.Future对象

使用线程池、进程池实现异步操作时用到的对象

```
import time
from concurrent.futures import Future
from concurrent.futures.thread import ThreadPoolExecutor
from concurrent.futures.process import ProcessPoolExecutor


def func(value):
    time.sleep(1)
    print(value)
    return 123


#创建线程池
pool = ThreadPoolExecutor(max_workers=5)

#创建进程池
# pool=ProcessPoolExecutor(max_workers=5)

#fut是线程池中一个func，参数i的结果
for i in range(10):
    fut = pool.submit(func, i)
    print(fut.result())
```

写代码时可能会存在交叉使用。例如：crm项目80%基于协程异步编程+mysql（不支持协程）【线程、进程做异步编程】

```
import time
import asyncio
import concurrent.futures


def func1():
    time.sleep(2)
    return 'SB'


async def main():
    loop = asyncio.get_running_loop()

    #1.run in the default loop's executor（默认threadpoolexecutor）
    #第一步：内部会先调用Threadpoolexecutor的submit方法去线程池中申请一个线程去执行func1函
数
    #返回一个concurrent.futures.Futures对象
    #第二步：调用asyncio.wrap_future方法将concurrent.futures.Future对象
```

```
    #包装为asyncio.Future对象
    #因为concurrent.futures.Future对象不支持await语法，所以需要包装
    fut = loop.run_in_executor(None, func1)
    #包装后的fut可以使用await
    result = await fut
    print('default thread pool', result)

    # 2.run in custom thread pool"
    # with concurrent.futures.ThreadPoolExecutor() as pool:
            #传入了pool作为新的循环上下文
    #     result = await loop.run_in_executor(pool, func1)
    #     print('custom thread pool', result)

    # 3.run in a custom process pool
    # with concurrent.futures.ProcessPoolExecutor() as pool:
    #     result = await loop.run_in_executor(pool, func1)
    #     print('custom process pool', result)


asyncio.run(main)
```

案例：

```
import asyncio
import requests


async def download_image(url):
    print('start', url)
    loop = asyncio.get_event_loop()
    #requests模块默认不支持异步操作，所以就使用线程池来配合实现
    future = loop.run_in_executor(None, requests.get, url)

    response = await future
    print('finished')
    file_name = url.rsplit('-')[-1]
    with open(file_name, mode='wb') as file_object:
        file_object.write(response.content)


if __name__ == '__main__':
    url_list = [
        1,
        2,
        3,
    ]

    tasks = [download_image(url) for url in url_list]

    loop = asyncio.get_event_loop()
    loop.run_until_complete(asyncio.wait(tasks))
```

3.7 异步迭代器

什么是异步迭代器

实现了aiter和anext方法的对象，anext必须返回一个awaitable对象，async for 会处理异步迭代器的anext方法并返回可等待对象，直到其引发stopasynciteration异常

什么是异步可迭代对象

可在async for语句中被使用的对象，必须通过它的aiter方法返回一个asynchronous iterator

```
import asyncio


class Reader(object):

    def __init__(self):
        self.count = 0

    #需要加async
    async def readline(self):
        self.count += 1
        if self.count == 100:
            return None
        return self.count

    def __aiter__(self):
        return self

    #需要加async
    async def __anext__(self):
        val = await self.readline()
        if val == None:
            raise StopAsyncIteration
        return val


async def func():
    obj = Reader()
    #async for必须在async的函数中使用
    async for item in obj:
        print(item)


asyncio.run(func())
```

3.8 异步上下文管理器

此种对象通过定义aenter和aexit方法来对async with语句中的环境进行控制

```
import asyncio
```

```
class AsyncContextManager:

    def __init__(self):
        self.conn = conn

    async def do_something(self):
        return 666

    async def __aenter__(self):
        self.conn = await asyncio.sleep(1)
        return self

    async def __aexit__(self):
        await asyncio.sleep(1)


obj = AsyncContextManager()


#async with必须嵌套在一个async函数中
async def func():
    async with obj as f:
        result = await f.do_somthing()
        print(result)


asyncio.run(func())
```

4.uvloop

是asyncio的事件循环的替代方案。uvloop的事件循环效率高于asyncio。

```
import asyncio
import uvloop

asyncio.set_event_loop_policy(uvloop.EventLoopPolicy())

#编写asyncio的代码与hi前代码一致

#内部的事件循环会自动变为uvloop

asyncio.run()
```

注意：一个asgi ---> uvicorn（django3 内部使用了这个，用uvloop写的）

5.实战案例

5.1 异步redis

在使用python代码操作redis时，链接/操作/断开都是网络io

示例1：

```
pip install aioredis


import asyncio
import aioredis

async def execute(address,password):
    #程序中的await本质是在等待io时让出资源，让其他的execute可以获得使用权
    print('start:',address)
    redis = await aioredis.create_redis(address,password)

    await redis.hmset_dict('car',key1=1,key2=2,key3=3)

    result=await redis.hgetall('car',encoding='utf-8')
    print(result)

    redis.close()

    await redis.wait_closed()

    print('finished',address)

asyncio.run(execute(address,password))
```

示例2：

```
import asyncio
import aioredis


async def execute(address, password):
    print('start:', address)
    redis = await aioredis.create_redis(address, password)

    await redis.hmset_dict('car', key1=1, key2=2, key3=3)

    result = await redis.hgetall('car', encoding='utf-8')
    print(result)

    redis.close()

    await redis.wait_closed()

    print('finished', address)


task_list = [
    execute(redis1, ps1),
    execute(redis2, ps2),
```

```
]

asyncio.run(asyncio.wait(task_lists))
```

5.2 异步MySQL

```
pip install aiomysql
```

示例1：

```
import asyncio
import aiomysql


async def execute():
    conn = await aiomysql.connect(
        host='111',
        port=3306,
        user='aaa',
        password='123',
        db='mysql',
    )

    cur = await conn.cursor()

    await cur.execute('SELECT HOST,User FROM user')

    result = await cur.fetchall()
    print(result)

    await cur.close()
    conn.close()


asyncio.run(execute())
```

示例2：

```
import asyncio
import aiomysql


async def execute(host, password):
    conn = await aiomysql.connect(
        host=host,
        port=3306,
        user='aaa',
        password=password,
        db='mysql',
    )

    cur = await conn.cursor()
```

```
    await cur.execute('SELECT HOST,User FROM user')

    result = await cur.fetchall()
    print(result)

    await cur.close()
    conn.close()


tasks_list = [
    execute('1', '1'),
    execute('2', '2'),
]

asyncio.run(asyncio.wait(tasks_list))
```

5.3 FastAPI 框架

```
pip install fastapi
pip install uvicorn #asgi内基于uvloop
```

示例:

```
import asyncio,aioredis

import uvicorn

from fastapi import FastAPI

app=FastAPI()

REDIS_POOL=aioredis.ConnectionPool('redis.....',password='asdf',minsize=1,maxsiz
e=10)


@app.get('/red')
async def red():
    print('request')
    await asyncio.sleep(3)
    conn=await REDIS_POOL.acquire()
    redis=Redis(conn)

    await redis.hmset_dict('car',key1=1,key2=2)

    result = await redis.hgetall('car',encoding='utf-8')

    REDIS_POOL.release(conn)

    return result

@app.get('/')
def index():
    return {'message':'hello world'}
```

```
if __name__='__main__':
    #luffy脚本名称，app，app装饰器
    uvicorn.run('luffy:app',host='127.0.0.1',port=5000,log_level='info')
```

5.4 异步爬虫

```
pip install aiohttp
```

示例：

```
import aiohttp
import asyncio


async def fetch(session, url):
    print('发送请求:', url)
    async with session.get(url, verify_ssl=False) as response:
        text = await response.text()
        print('得到结果：', url, len(text))


async def main():
    async with aiohttp.ClientSession() as session:
        url_list = ['https://python.org', 'https://www.baidu.com']
        tasks = [asyncio.create_task(fetch(session, url)) for url in url_list]

        await asyncio.wait(tasks)


if __name__ == '__main__':
    asyncio.run(main())
```

总结

最大的意义：通过一个线程利用其IO等待时间去做一些其他事情。