Python黑魔法 --- 异步IO（ asyncio）协程

人世间 (/u/5qrPPM) ＋关注
2017.01.17 11:12* 字数 3396 阅读 17773 评论 46 喜欢 79 赞赏 2
(/u/5qrPPM)

python asyncio

网络模型有很多中，为了实现高并发也有很多方案，多线程，多进程。无论多线程和多进程，IO的调度更多取决于系统，而协程的方式，调度来自用户，用户可以在函数中yield一个状态。使用协程可以实现高效的并发任务。Python的在3.4中引入了协程的概念，可是这个还是以生成器对象为基础，3.5则确定了协程的语法。下面将简单介绍asyncio的使用。实现协程的不仅仅是asyncio，tornado和gevent都实现了类似的功能。

- event_loop 事件循环：程序开启一个无限的循环，程序员会把一些函数注册到事件循环上。当满足事件发生的时候，调用相应的协程函数。

- coroutine 协程：协程对象，指一个使用async关键字定义的函数，它的调用不会立即执行函数，而是会返回一个协程对象。协程对象需要注册到事件循环，由事件循环调用。

- task 任务：一个协程对象就是一个原生可以挂起的函数，任务则是对协程进一步封装，其中包含任务的各种状态。

- future： 代表将来执行或没有执行的任务的结果。它和task上没有本质的区别

- async/await 关键字：python3.5 用于定义协程的关键字，async定义一个协程，await用于挂起阻塞的异步调用接口。

上述的概念单独拎出来都不好懂，比较他们之间是相互联系，一起工作。下面看例子，再回溯上述概念，更利于理解。

定义一个协程

定义一个协程很简单，使用async关键字，就像定义普通函数一样：

```
import time
import asyncio

now = lambda : time.time()

async def do_some_work(x):
    print('Waiting: ', x)

start = now()

coroutine = do_some_work(2)

loop = asyncio.get_event_loop()
loop.run_until_complete(coroutine)

print('TIME: ', now() - start)
```

通过async关键字定义一个协程（coroutine），协程也是一种对象。协程不能直接运行，需要把协程加入到事件循环（loop），由后者在适当的时候调用协程。
`asyncio.get_event_loop` 方法可以创建一个事件循环，然后使用 `run_until_complete` 将协程注册到事件循环，并启动事件循环。因为本例只有一个协程，于是可以看见如下输出：

```
Waiting:  2
TIME:  0.0004658699035644531
```

创建一个task

协程对象不能直接运行，在注册事件循环的时候，其实是run_until_complete方法将协程包装成为了一个任务（task）对象。所谓task对象是Future类的子类。保存了协程运行后的状态，用于未来获取协程的结果。

```
import asyncio
import time

now = lambda : time.time()

async def do_some_work(x):
    print('Waiting: ', x)

start = now()

coroutine = do_some_work(2)
loop = asyncio.get_event_loop()
# task = asyncio.ensure_future(coroutine)
task = loop.create_task(coroutine)
print(task)
loop.run_until_complete(task)
print(task)
print('TIME: ', now() - start)
```

可以看到输出结果为：

```
<Task pending coro=<do_some_work() running at /Users/ghost/Rsj217/python3.6/async/as
Waiting:  2
<Task finished coro=<do_some_work() done, defined at /Users/ghost/Rsj217/python3.6/a
TIME:  0.0003490447998046875
```

创建task后，task在加入事件循环之前是pending状态，因为do_some_work中没有耗时的阻塞操作，task很快就执行完毕了。后面打印的finished状态。

asyncio.ensure_future(coroutine) 和 loop.create_task(coroutine)都可以创建一个task，run_until_complete的参数是一个futrue对象。当传入一个协程，其内部会自动封装成task，task是Future的子类。 `isinstance(task, asyncio.Future)` 将会输出True。

绑定回调

绑定回调，在task执行完毕的时候可以获取执行的结果，回调的最后一个参数是future对象，通过该对象可以获取协程返回值。如果回调需要多个参数，可以通过偏函数导入。

```
import time
import asyncio

now = lambda : time.time()

async def do_some_work(x):
    print('Waiting: ', x)
    return 'Done after {}s'.format(x)

def callback(future):
    print('Callback: ', future.result())

start = now()

coroutine = do_some_work(2)
loop = asyncio.get_event_loop()
task = asyncio.ensure_future(coroutine)
task.add_done_callback(callback)
loop.run_until_complete(task)

print('TIME: ', now() - start)
```

```
def callback(t, future):
    print('Callback:', t, future.result())

task.add_done_callback(functools.partial(callback, 2))
```

可以看到，coroutine执行结束时候会调用回调函数。并通过参数future获取协程执行的结果。我们创建的task和回调里的future对象，实际上是同一个对象。

future 与 result

回调一直是很多异步编程的恶梦，程序员更喜欢使用同步的编写方式写异步代码，以避免回调的恶梦。回调中我们使用了future对象的result方法。前面不绑定回调的例子中，我们可以看到task有fiinished状态。在那个时候，可以直接读取task的result方法。

```
async def do_some_work(x):
    print('Waiting {}'.format(x))
    return 'Done after {}s'.format(x)

start = now()

coroutine = do_some_work(2)
loop = asyncio.get_event_loop()
task = asyncio.ensure_future(coroutine)
loop.run_until_complete(task)

print('Task ret: {}'.format(task.result()))
print('TIME: {}'.format(now() - start))
```


可以看到输出的结果：

```
Waiting:  2
Task ret:  Done after 2s
TIME:  0.0003650188446044922
```

阻塞和await

使用async可以定义协程对象，使用await可以针对耗时的操作进行挂起，就像生成器里的yield一样，函数让出控制权。协程遇到await，事件循环将会挂起该协程，执行别的协程，直到其他的协程也挂起或者执行完毕，再进行下一个协程的执行。

耗时的操作一般是一些IO操作，例如网络请求，文件读取等。我们使用asyncio.sleep函数来模拟IO操作。协程的目的也是让这些IO操作异步化。

```
import asyncio
import time

now = lambda: time.time()

async def do_some_work(x):
    print('Waiting: ', x)
    await asyncio.sleep(x)
    return 'Done after {}s'.format(x)

start = now()

coroutine = do_some_work(2)
loop = asyncio.get_event_loop()
task = asyncio.ensure_future(coroutine)
loop.run_until_complete(task)

print('Task ret: ', task.result())
print('TIME: ', now() - start)
```

在 sleep的时候，使用await让出控制权。即当遇到阻塞调用的函数的时候，使用await方法将协程的控制权让出，以便loop调用其他的协程。现在我们的例子就用耗时的阻塞操作了。

并发和并行

并发和并行一直是容易混淆的概念。并发通常指有多个任务需要同时进行，并行则是同一时刻有多个任务执行。用上课来举例就是，并发情况下是一个老师在同一时间段辅助不同的人功课。并行则是好几个老师分别同时辅助多个学生功课。简而言之就是一个人

同时吃三个馒头还是三个人同时分别吃一个的情况，吃一个馒头算一个任务。

asyncio实现并发，就需要多个协程来完成任务，每当有任务阻塞的时候就await，然后其他协程继续工作。创建多个协程的列表，然后将这些协程注册到事件循环中。


```
import asyncio

import time

now = lambda: time.time()

async def do_some_work(x):
    print('Waiting: ', x)

    await asyncio.sleep(x)
    return 'Done after {}s'.format(x)

start = now()

coroutine1 = do_some_work(1)
coroutine2 = do_some_work(2)
coroutine3 = do_some_work(4)

tasks = [
    asyncio.ensure_future(coroutine1),
    asyncio.ensure_future(coroutine2),
    asyncio.ensure_future(coroutine3)
]

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

for task in tasks:
    print('Task ret: ', task.result())

print('TIME: ', now() - start)
```

结果如下

```
Waiting:  1
Waiting:  2
Waiting:  4
Task ret:  Done after 1s
Task ret:  Done after 2s
Task ret:  Done after 4s
TIME:  4.003541946411133
```

总时间为4s左右。4s的阻塞时间，足够前面两个协程执行完毕。如果是同步顺序的任务，那么至少需要7s。此时我们使用了aysncio实现了并发。asyncio.wait(tasks) 也可以使用 asyncio.gather(*tasks) ,前者接受一个task列表，后者接收一堆task。

协程嵌套

使用async可以定义协程，协程用于耗时的io操作，我们也可以封装更多的io操作过程，这样就实现了嵌套的协程，即一个协程中await了另外一个协程，如此连接起来。

```
import asyncio

import time

now = lambda: time.time()

async def do_some_work(x):
    print('Waiting: ', x)

    await asyncio.sleep(x)
    return 'Done after {}s'.format(x)

async def main():
    coroutine1 = do_some_work(1)
    coroutine2 = do_some_work(2)
    coroutine3 = do_some_work(4)

    tasks = [
        asyncio.ensure_future(coroutine1),
        asyncio.ensure_future(coroutine2),
        asyncio.ensure_future(coroutine3)
    ]

    dones, pendings = await asyncio.wait(tasks)

    for task in dones:
        print('Task ret: ', task.result())

start = now()

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

print('TIME: ', now() - start)
```


如果使用的是 asyncio.gather创建协程对象，那么await的返回值就是协程运行的结果。

```
    results = await asyncio.gather(*tasks)

    for result in results:
        print('Task ret: ', result)
```

不在main协程函数里处理结果，直接返回await的内容，那么最外层的run_until_complete将会返回main协程的结果。

```
async def main():
    coroutine1 = do_some_work(1)
    coroutine2 = do_some_work(2)
    coroutine3 = do_some_work(2)

    tasks = [
        asyncio.ensure_future(coroutine1),
        asyncio.ensure_future(coroutine2),
        asyncio.ensure_future(coroutine3)
    ]

    return await asyncio.gather(*tasks)

start = now()

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

for result in results:
    print('Task ret: ', result)
```


或者返回使用asyncio.wait方式挂起协程。

```
async def main():
    coroutine1 = do_some_work(1)
    coroutine2 = do_some_work(2)
    coroutine3 = do_some_work(4)

    tasks = [
        asyncio.ensure_future(coroutine1),
        asyncio.ensure_future(coroutine2),
        asyncio.ensure_future(coroutine3)
    ]

    return await asyncio.wait(tasks)

start = now()

loop = asyncio.get_event_loop()
done, pending = loop.run_until_complete(main())

for task in done:
    print('Task ret: ', task.result())
```

也可以使用asyncio的as_completed方法

```
async def main():
    coroutine1 = do_some_work(1)
    coroutine2 = do_some_work(2)
    coroutine3 = do_some_work(4)

    tasks = [
        asyncio.ensure_future(coroutine1),
        asyncio.ensure_future(coroutine2),
        asyncio.ensure_future(coroutine3)
    ]
    for task in asyncio.as_completed(tasks):
        result = await task
        print('Task ret: {}'.format(result))

start = now()

loop = asyncio.get_event_loop()
done = loop.run_until_complete(main())
print('TIME: ', now() - start)
```


由此可见，协程的调用和组合十分灵活，尤其是对于结果的处理，如何返回，如何挂起，需要逐渐积累经验和前瞻的设计。

协程停止

上面见识了协程的几种常用的用法，都是协程围绕着事件循环进行的操作。future对象有几个状态：

- Pending
- Running
- Done
- Cancelled

创建future的时候，task为pending，事件循环调用执行的时候当然就是running，调用完毕自然就是done，如果需要停止事件循环，就需要先把task取消。可以使用asyncio.Task获取事件循环的task

```
import asyncio

import time

now = lambda: time.time()

async def do_some_work(x):
    print('Waiting: ', x)

    await asyncio.sleep(x)
    return 'Done after {}s'.format(x)

coroutine1 = do_some_work(1)
coroutine2 = do_some_work(2)
coroutine3 = do_some_work(2)

tasks = [
    asyncio.ensure_future(coroutine1),
    asyncio.ensure_future(coroutine2),
    asyncio.ensure_future(coroutine3)
]

start = now()

loop = asyncio.get_event_loop()
try:
    loop.run_until_complete(asyncio.wait(tasks))
except KeyboardInterrupt as e:
    print(asyncio.Task.all_tasks())
    for task in asyncio.Task.all_tasks():
        print(task.cancel())
    loop.stop()
    loop.run_forever()
finally:
    loop.close()

print('TIME: ', now() - start)
```


启动事件循环之后，马上ctrl+c，会触发run_until_complete的执行异常KeyBorardInterrupt。然后通过循环asyncio.Task取消future。可以看到输出如下：

```
Waiting:  1
Waiting:  2
Waiting:  2
{<Task pending coro=<do_some_work() running at /Users/ghost/Rsj217/python3.6/async/a
True
True
True
True
TIME:  0.8858370780944824
```

True表示cannel成功，loop stop之后还需要再次开启事件循环，最后在close，不然还会抛出异常：

```
Task was destroyed but it is pending!
task: <Task pending coro=<do_some_work() done,
```

循环task，逐个cancel是一种方案，可是正如上面我们把task的列表封装在main函数中，main函数外进行事件循环的调用。这个时候，main相当于最外出的一个task，那么处理包装的main函数即可。

```
import asyncio

import time

now = lambda: time.time()

async def do_some_work(x):
    print('Waiting: ', x)

    await asyncio.sleep(x)
    return 'Done after {}s'.format(x)

async def main():
    coroutine1 = do_some_work(1)
    coroutine2 = do_some_work(2)
    coroutine3 = do_some_work(2)

    tasks = [
        asyncio.ensure_future(coroutine1),
        asyncio.ensure_future(coroutine2),
        asyncio.ensure_future(coroutine3)
    ]
    done, pending = await asyncio.wait(tasks)
    for task in done:
        print('Task ret: ', task.result())

start = now()

loop = asyncio.get_event_loop()
task = asyncio.ensure_future(main())
try:
    loop.run_until_complete(task)
except KeyboardInterrupt as e:
    print(asyncio.Task.all_tasks())
    print(asyncio.gather(*asyncio.Task.all_tasks()).cancel())
    loop.stop()
    loop.run_forever()
finally:
    loop.close()
```

不同线程的事件循环

很多时候，我们的事件循环用于注册协程，而有的协程需要动态的添加到事件循环中。一个简单的方式就是使用多线程。当前线程创建一个事件循环，然后在新建一个线程，在新线程中启动事件循环。当前线程不会被block。

```
from threading import Thread

def start_loop(loop):
    asyncio.set_event_loop(loop)
    loop.run_forever()

def more_work(x):
    print('More work {}'.format(x))
    time.sleep(x)
    print('Finished more work {}'.format(x))

start = now()
new_loop = asyncio.new_event_loop()
t = Thread(target=start_loop, args=(new_loop,))
t.start()
print('TIME: {}'.format(time.time() - start))

new_loop.call_soon_threadsafe(more_work, 6)
new_loop.call_soon_threadsafe(more_work, 3)
```

启动上述代码之后，当前线程不会被block，新线程中会按照顺序执行call_soon_threadsafe方法注册的more_work方法，后者因为time.sleep操作是同步阻塞的，因此运行完毕more_work需要大致6 + 3

新线程协程

```
def start_loop(loop):
    asyncio.set_event_loop(loop)
    loop.run_forever()

async def do_some_work(x):
    print('Waiting {}'.format(x))
    await asyncio.sleep(x)
    print('Done after {}s'.format(x))

def more_work(x):
    print('More work {}'.format(x))
    time.sleep(x)
    print('Finished more work {}'.format(x))

start = now()
new_loop = asyncio.new_event_loop()
t = Thread(target=start_loop, args=(new_loop,))
t.start()
print('TIME: {}'.format(time.time() - start))

asyncio.run_coroutine_threadsafe(do_some_work(6), new_loop)
asyncio.run_coroutine_threadsafe(do_some_work(4), new_loop)
```

上述的例子，主线程中创建一个new_loop，然后在另外的子线程中开启一个无限事件循环。主线程通过run_coroutine_threadsafe新注册协程对象。这样就能在子线程中进行事件循环的并发操作，同时主线程又不会被block。一共执行的时间大概在6s左右。

master-worker主从模式

对于并发任务，通常是用生成消费模型，对队列的处理可以使用类似master-worker的方式，master主要用户获取队列的msg，worker用户处理消息。

为了简单起见，并且协程更适合单线程的方式，我们的主线程用来监听队列，子线程用于处理队列。这里使用redis的队列。主线程中有一个是无限循环，用户消费队列。

```
    while True:
        task = rcon.rpop("queue")
        if not task:
            time.sleep(1)
            continue
        asyncio.run_coroutine_threadsafe(do_some_work(int(task)), new_loop)
```

给队列添加一些数据：

```
127.0.0.1:6379[3]> lpush queue 2
(integer) 1
127.0.0.1:6379[3]> lpush queue 5
(integer) 1
127.0.0.1:6379[3]> lpush queue 1
(integer) 1
127.0.0.1:6379[3]> lpush queue 1
```

可以看见输出：

```
Waiting  2
Done 2
Waiting  5
Waiting  1
Done 1
Waiting  1
Done 1
Done 5
```


我们发起了一个耗时5s的操作，然后又发起了连个1s的操作，可以看见子线程并发的执行了这几个任务，其中5s awati的时候，相继执行了1s的两个任务。

停止子线程

如果一切正常，那么上面的例子很完美。可是，需要停止程序，直接ctrl+c，会抛出KeyboardInterrupt错误，我们修改一下主循环：

```
try:
    while True:
        task = rcon.rpop("queue")
        if not task:
            time.sleep(1)
            continue
        asyncio.run_coroutine_threadsafe(do_some_work(int(task)), new_loop)
except KeyboardInterrupt as e:
    print(e)
    new_loop.stop()
```

可是实际上并不好使，虽然主线程try了KeyboardInterrupt异常，但是子线程并没有退出，为了解决这个问题，可以设置子线程为守护线程，这样当主线程结束的时候，子线程也随机退出。

```
new_loop = asyncio.new_event_loop()
t = Thread(target=start_loop, args=(new_loop,))
t.setDaemon(True)    # 设置子线程为守护线程
t.start()

try:
    while True:
        # print('start rpop')
        task = rcon.rpop("queue")
        if not task:
            time.sleep(1)
            continue
        asyncio.run_coroutine_threadsafe(do_some_work(int(task)), new_loop)
except KeyboardInterrupt as e:
    print(e)
    new_loop.stop()
```

线程停止程序的时候，主线程退出后，子线程也随机退出才了，并且停止了子线程的协程任务。

aiohttp

在消费队列的时候，我们使用asyncio的sleep用于模拟耗时的io操作。以前有一个短信服务，需要在协程中请求远程的短信api，此时需要是需要使用aiohttp进行异步的http请求。大致代码如下：

server.py

```
import time
from flask import Flask, request

app = Flask(__name__)

@app.route('/<int:x>')
def index(x):
    time.sleep(x)
    return "{} It works".format(x)

@app.route('/error')
def error():
    time.sleep(3)
    return "error!"

if __name__ == '__main__':
    app.run(debug=True)
```

`/` 接口表示短信接口， `/error` 表示请求 `/` 失败之后的报警。

async-custoimer.py

```
import time
import asyncio
from threading import Thread
import redis
import aiohttp

def get_redis():
    connection_pool = redis.ConnectionPool(host='127.0.0.1', db=3)
    return redis.Redis(connection_pool=connection_pool)

rcon = get_redis()

def start_loop(loop):
    asyncio.set_event_loop(loop)
    loop.run_forever()

async def fetch(url):
    async with aiohttp.ClientSession() as session:
        async with session.get(url) as resp:
            print(resp.status)
            return await resp.text()

async def do_some_work(x):
    print('Waiting ', x)
    try:
        ret = await fetch(url='http://127.0.0.1:5000/{}'.format(x))
        print(ret)
    except Exception as e:
        try:
            print(await fetch(url='http://127.0.0.1:5000/error'))
        except Exception as e:
            print(e)
    else:
        print('Done {}'.format(x))

new_loop = asyncio.new_event_loop()
t = Thread(target=start_loop, args=(new_loop,))
t.setDaemon(True)
t.start()

try:
    while True:
        task = rcon.rpop("queue")
        if not task:
            time.sleep(1)
            continue
        asyncio.run_coroutine_threadsafe(do_some_work(int(task)), new_loop)
except Exception as e:
    print('error')
    new_loop.stop()
finally:
    pass
```

有一个问题需要注意，我们在fetch的时候try了异常，如果没有try这个异常，即使发生了异常，子线程的事件循环也不会退出。主线程也不会退出，暂时没找到办法可以把子线程的异常raise传播到主线程。（如果谁找到了比较好的方式，希望可以带带我）。

对于redis的消费，还有一个block的方法：

```
try:
    while True:
        _, task = rcon.brpop("queue")
        asyncio.run_coroutine_threadsafe(do_some_work(int(task)), new_loop)
except Exception as e:
    print('error', e)
    new_loop.stop()
finally:
    pass
```


使用 brpop方法，会block住task，如果主线程有消息，才会消费。测试了一下，似乎brpop的方式更适合这种队列消费的模型。

```
127.0.0.1:6379[3]> lpush queue 5
(integer) 1
127.0.0.1:6379[3]> lpush queue 1
(integer) 1
127.0.0.1:6379[3]> lpush queue 1
```

可以看到结果

```
Waiting  5
Waiting  1
Waiting  1
200
1 It works
Done 1
200
1 It works
Done 1
200
5 It works
Done 5
```

协程消费

主线程用于监听队列，然后子线程的做事件循环的worker是一种方式。还有一种方式实现这种类似master-worker的方案。即把监听队列的无限循环逻辑一道协程中。程序初始化就创建若干个协程，实现类似并行的效果。

```
import time
import asyncio
import redis

now = lambda : time.time()

def get_redis():
    connection_pool = redis.ConnectionPool(host='127.0.0.1', db=3)
    return redis.Redis(connection_pool=connection_pool)

rcon = get_redis()

async def worker():
    print('Start worker')

    while True:
        start = now()
        task = rcon.rpop("queue")
        if not task:
            await asyncio.sleep(1)
            continue
        print('Wait ', int(task))
        await asyncio.sleep(int(task))
        print('Done ', task, now() - start)

def main():
    asyncio.ensure_future(worker())
    asyncio.ensure_future(worker())

    loop = asyncio.get_event_loop()
    try:
        loop.run_forever()
    except KeyboardInterrupt as e:
        print(asyncio.gather(*asyncio.Task.all_tasks()).cancel())
        loop.stop()
        loop.run_forever()
    finally:
        loop.close()

if __name__ == '__main__':
    main()
```


这样做就可以多多启动几个worker来监听队列。一样可以到达效果。

总结

上述简单的介绍了asyncio的用法，主要是理解事件循环，协程和任务，future的关系。异步编程不同于常见的同步编程，设计程序的执行流的时候，需要特别的注意。毕竟这和以往的编码经验有点不一样。可是仔细想想，我们平时处事的时候，大脑会自然而然的实现异步协程。比如等待煮茶的时候，可以多写几行代码。

相关代码文件的Gist (https://link.jianshu.com?t=https://gist.github.com/rsj217/f6b9ddadebfa4a19c14c6d32b1e961f1)

参考：Threaded Asynchronous Magic and How to Wield It (https://link.jianshu.com?t=https://hackernoon.com/threaded-asynchronous-magic-and-how-to-wield-it-bba9ed602c32#.71h32j2ir)