asyncio

C-2 创建: 张林伟, 最后修改: 张林伟 2018-07-28 10:53

asyncio 从 3.4 开始成为 Python 生态系统的一部分，利用 asyncio 能轻松地编写利用协程的单线程并发程序。

概念

- 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。

定义协程

使用async关键字定义协程（coroutine）

代码块 Python

```python
async def MyCoroutine():
    print("Hello, world")
```

协程也是一种对象，它不能直接运行，需要把协程加入到事件循环（event_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。

代码块 Python

```python
import asyncio
loop = asyncio.get_event_loop()
loop.run_until_complete(MyCoroutine())
```

创建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。

代码块 Python

```python
import asyncio
import time

now = lambda : time.time()

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

start = now()

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

```
<Task pending coro=<MyCoroutine() running at demo4.py:6>>
Waiting 2
<Task finished coro=<MyCoroutine() done, defined at demo4.py:6> result=None>
TIME:  0.0004558563232421875
```

task在加入事件循环（event_loop）之前，处于pending状态，加入到event_loop并执行完毕后，处于finished状态。

两种方法创建task

- loop.create_task
- ansyncio.ensure_future

```
    # Method scheduling a coroutine object: create a task.

    def create_task(self, coro):
        raise NotImplementedError
```

```
def ensure_future(coro_or_future, *, loop=None):
    """Wrap a coroutine or an awaitable in a future.

    If the argument is a Future, it is returned directly.
    """
    if coroutines.iscoroutine(coro_or_future):
        if loop is None:
            loop = events.get_event_loop()
        task = loop.create_task(coro_or_future)
        if task._source_traceback:
            del task._source_traceback[-1]
        return task
    elif futures.isfuture(coro_or_future):
        if loop is not None and loop is not futures._get_loop(coro_or_future):
            raise ValueError('loop argument must agree with Future')
        return coro_or_future
    elif inspect.isawaitable(coro_or_future):
        return ensure_future(_wrap_awaitable(coro_or_future), loop=loop)
    else:
        raise TypeError('An asyncio.Future, a coroutine or an awaitable is '
                        'required')
```

绑定回调

```
import time
import asyncio

now = lambda _: time.time()

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

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

start = now()

loop = asyncio.get_event_loop()
task = asyncio.ensure_future(MyCoroutine(3))
task.add_done_callback(callback)
loop.run_until_complete(task)

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

使用task对象的add_done_callback方法为task添加回调函数。

打印结果：

```
Waiting:  3
Callback:  Done after 3s
Task ret:  Done after 3s
TIME:  0.0009479522705078125
```

阻塞和await

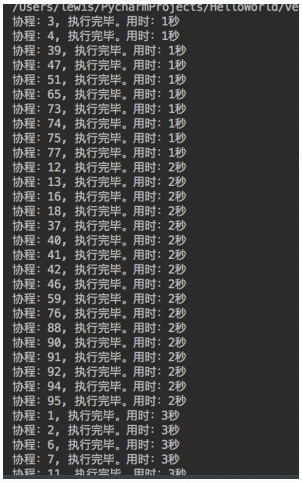
当执行一些耗时操作，使用await将此协程挂起，执行别的协程，直到其他的协程也挂起或者执行完毕，再进行下一个协程的执行。

代码块

```
import asyncio
import random

async def MyCoroutine(id):
    process_time = random.randint(1, 5)
    await asyncio.sleep(process_time)
    print('协程：{}，执行完毕。用时：{}秒'.format(id, process_time))

async def main():
    tasks = [asyncio.ensure_future(MyCoroutine(i)) for i in range(100)]
    await asyncio.gather(*tasks)

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

```
17        loop.close()
```

运行结果



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。

参考资料

https://www.jianshu.com/p/b5e347b3a17c

https://mp.weixin.qq.com/s/KWHjkZMzVnetuNwzYopWuw