asyncio协程

Python3对异步的支持：

asyncio是Python3.4版本引入的标准库，直接内置了对异步IO的支持.

关于asyncio的一些关键字的说明：

1. async/await是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.
   协程遇到await事件循环将会挂起该协程，执行别的协程；耗时的操作一般是一些IO操作，如网络请求 文件读取等；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.

2. coroutine协程对象，通过async关键字定义的函数，它的调用不会立即执行函数，而是返回一个协程对象，它需要注册到事件循环，由事件循环调用.

3. 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(协程)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.

4. future对象：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.

5. 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. 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.

In [ ]:
```
#示例代码:
async def f1(x):
    print('this is %s' %x)
    await ayncio.sleep(x)
    return 'task done! %s' %x

c1 = f1(1)
c2 = f1(2)
c3 = f1(3)
```

await asyncio.sleep(2)，模拟阻塞2秒，由于属于异步io操作，所以传统的time.sleep(2)的调用会导致同步的等待2秒，比如你使用10个协程去完成异步操作，time.sleep会耗掉10*2秒时间进行操作，使用await asyncio.sleep（2）会耗掉2秒多的时间操作，所以使用了time.sleep就无法享受到异步的好处,然后我们开始主函数里面进行调用.

In [ ]:
```
import asyncio
loop = asyncio.get_event_loop()
tasks = [loop.create_task(c1), loop.create_task(c2), loop.create_task(c3)]
print(tasks)
#可见任务的状态是pending
```

把任务放到事件循环中

In [ ]:
```
for t in tasks:
    loop.run_until_complete(t)
```

绑定回调函数：

异步IO的实现原理，就是在IO高的地方将代码挂起，等IO结束后再继续执行. 在绝大部分时候，我们后续的代码的执行是需要依赖IO的返回值的，这就要用到回调了.

回调的实现有两种：一种是绝大部分程序员喜欢的，利用的同步编程实现的回调. 这就要求我们要能够有办法取得协程的await的返回值. 获取每个任务的返回值，可以通过task.result()来获取.

In [ ]:
```
for t in tasks:
    print(t.result())
```

完整代码：

In [1]:
```
#示例代码:
import asyncio
import time

async def f1(x):
    print('this is %s' %x)
    await asyncio.sleep(x)
    return 'task done! %s' %x

loop = asyncio.get_event_loop()

c1 = f1(1)
c2 = f1(2)
c3 = f1(3)

tasks = [loop.create_task(c1), loop.create_task(c2), loop.create_task(c3)]
print(tasks)

beg = time.time()

for t in tasks:
    loop.run_until_complete(t)

for t in tasks:
    print(t.result())

end= time.time()
print(end-beg)
```

输出:

```
[<Task pending coro=<f1() running at demo12.py:4>>, <Task pending coro=<f1() running at demo12.py:4>>, <Task pending coro=<f1() running at demo12.py:4>>]
this is 1
this is 2
this is 3
task done! 1
task done! 2
task done! 3
3.0019843578338623
```

也可以通过为协程添加回调函数的方式来获取协程任务的返回值.

示例代码：

In [ ]:
```
[<Task pending coro=<f1() running at demo1.py:7>>, <Task pending coro=<f1() running at demo1.py:7>>, <Task pending coro=<f1() running at demo1.py:7>>]
this is 1
this is 2
this is 3
this is callback task done! 1
this is callback task done! 2
this is callback task done! 3
3.001849889755249
[root@nfs coding]#
[root@nfs coding]#
[root@nfs coding]#
[root@nfs coding]#   clear
[root@nfs coding]# ls
demo1.py
[root@nfs coding]# cat demo1.py
import asyncio
import time

def callback(x):
    print('this is callback %s' %x.result())

async def f1(x):
    print('this is %s' %x)
    await asyncio.sleep(x)
    return 'task done! %s' %x

loop = asyncio.get_event_loop()

c1 = f1(1)
c2 = f1(2)
c3 = f1(3)

tasks = [loop.create_task(c1), loop.create_task(c2), loop.create_task(c3)]
print(tasks)

beg = time.time()

for t in tasks:
    t.add_done_callback(callback)

for t in tasks:
    loop.run_until_complete(t)

end= time.time()
print(end-beg)
```

输出结果:

```
[<Task pending coro=<f1() running at demo1.py:7>>, <Task pending coro=<f1() running at demo1.py:7>>, <Task pending coro=<f1() running at demo1.py:7>>]
this is 1
this is 2
this is 3
this is callback task done! 1
this is callback task done! 2
this is callback task done! 3
3.0023529529571533
```

动态添加协程任务:

示例代码(主线程是同步的):

In [ ]:
```
import time
import asyncio
from queue import Queue
from threading import Thread


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


def do_sleep(x, queue, msg=""):
    time.sleep(x)
    queue.put(msg)


queue = Queue()

loop = asyncio.new_event_loop()

t = Thread(target=start_loop, args=(loop, ))
t.start()

beg = time.time()

#动态添加两个协程

loop.call_soon_threadsafe(do_sleep, 1, queue, "first")
loop.call_soon_threadsafe(do_sleep, 2, queue, "second")

while 1:
    msg = queue.get()
    print("%s协程运行完毕.." %msg)
    end = time.time()
    print(end - beg)
```

```
first协程运行完毕..
1.0045161247253418
second协程运行完毕..
3.0078980922698975
```

示例代码(主线程是异步的):

In [ ]:
```
import time
import asyncio
from queue import Queue
from threading import Thread


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


async def do_sleep(x, queue, msg=""):
    await asyncio.sleep(x)
    queue.put(msg)


queue = Queue()

loop = asyncio.new_event_loop()

t = Thread(target=start_loop, args=(loop, ))
t.start()

beg = time.time()

#动态添加两个协程

asyncio.run_coroutine_threadsafe(do_sleep(1, queue, "first"), loop)
asyncio.run_coroutine_threadsafe(do_sleep(2, queue, "second"), loop)

while 1:
    msg = queue.get()
    print("%s协程运行完毕.." %msg)

    end = time.time()
    print(end - beg)
```

```
first协程运行完毕..
1.0035889148712158
second协程运行完毕..
2.004390001296997
```

await与yield对比

1.yield from后面可接可迭代对象; 也可接future对象/协程对象;
2.await 后面必须要接future对象/协程对象

示例代码:

In [ ]:
```
import asyncio
import time

async def f1(x):
    print('this is %s' %x)
    await asyncio.sleep(x)
    return 'task done! %s' %x

loop = asyncio.get_event_loop()

c1 = f1(1)
c2 = f1(2)
c3 = f1(3)

tasks = [loop.create_task(c1), loop.create_task(c2), loop.create_task(c3)]
print(tasks)

beg = time.time()

for t in tasks:
    loop.run_until_complete(t)

for t in tasks:
    print(t.result())

end= time.time()
print(end-beg)
```

将以上代码替换为如下样式, 得出的结果也是相同的

In [ ]:
```
import asyncio
import time

def f1(x):
    print('this is %s' %x)
    yield from asyncio.sleep(x)
    return 'task done! %s' %x

loop = asyncio.get_event_loop()

c1 = f1(1)
c2 = f1(2)
c3 = f1(3)

tasks = [loop.create_task(c1), loop.create_task(c2), loop.create_task(c3)]
print(tasks)

beg = time.time()

for t in tasks:
    loop.run_until_complete(t)

for t in tasks:
    print(t.result())

end= time.time()
print(end-beg)
```

协程嵌套

In [ ]:
```
import asyncio
import time
# 用于内部的协程函数
async def do_some_work(x):
    print('Waiting: ', x)
    await asyncio.sleep(x)
    return 'Done after {}s'.format(x)

# 外部的协程函数
async def main():
    # 创建三个协程对象
    coroutine1 = do_some_work(1)
    coroutine2 = do_some_work(2)
    coroutine3 = do_some_work(4)

    # 将协程转为task，并组成list
    tasks = [
        asyncio.ensure_future(coroutine1),
        asyncio.ensure_future(coroutine2),
        asyncio.ensure_future(coroutine3)
    ]

    # await 一个task列表（协程）
    # dones：表示已经完成的任务
    # pendings：表示未完成的任务
    dones, pendings = await asyncio.wait(tasks)

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

beg = time.time()
loop = asyncio.get_event_loop()
loop.run_until_complete(main())
end = time.time()
print(end - beg)
```