

协程和异步 IO

协程的概念

协程，又称微线程，纤程。英文名 **Coroutine**，是一种用户态的轻量级线程。

子程序，或者称为函数，在所有语言中都是层级调用，比如 A 调用 B，B 在执行过程中又调用了 C，C 执行完毕返回，B 执行完毕返回，最后是 A 执行完毕。所以子程序调用是通过栈实现的，一个线程就是执行一个子程序。子程序调用总是一个入口，一次返回，调用顺序是明确的。而协程的调用和子程序不同。

线程是系统级别的它们由操作系统调度，而协程则是程序级别的由程序根据需要自己调度。在一个线程中会有很多函数，我们把这些函数称为子程序，在子程序执行过程中可以中断去执行别的子程序，而别的子程序也可以中断回来继续执行之前的子程序，这个过程就称为协程。也就是说在同一线程内一段代码在执行过程中会中断然后跳转执行别的代码，接着在之前中断的地方继续开始执行。

协程拥有自己的寄存器上下文和栈。协程调度切换时，将寄存器上下文和栈保存到其他地方，在切回来的时候，恢复先前保存的寄存器上下文和栈。因此：协程能保留上一次调用时的状态（即所有局部状态的一个特定组合），每次过程重入时，就相当于进入上一次调用的状态，换种说法：进入上一次离开时所处逻辑流的位置。

【示例】代码描述协程

```
def A():  
    print('1')  
    print('2')  
    print('3')  
  
def B():  
    print('x')  
    print('y')  
    print('z')
```

由协程执行，在执行 A 的过程中，可以随时中断，去执行 B，B 也可能在执行过程中中断再去执行 A，可能的结果是：

```
1  
2  
x  
y  
3
```

Z

但是在 A 中是没有调用 B 的，所以协程的调用比函数调用理解起来要难一些，看起来 A、B 的执行有点像多线程，但协程的特点在于是一个线程执行，那和多线程比，协程有何优势？

协程的优点：

(1) 无需线程上下文切换的开销，协程避免了无意义的调度，由此可以提高性能（但也因此，程序员必须自己承担调度的责任，同时，协程也失去了标准线程使用多 CPU 的能力）

(2) 无需原子操作锁定及同步的开销

(3) 方便切换控制流，简化编程模型

(4) 高并发+高扩展性+低成本：一个 CPU 支持上万的协程都不是问题。所以很适合用于高并发处理。

协程的缺点：

(1) 无法利用多核资源：协程的本质是个单线程,它不能同时将单个 CPU 的多个核用上,协程需要和进程配合才能运行在多 CPU 上.当然我们日常所编写的绝大部分应用都没有这个必要，除非是 cpu 密集型应用。

(2) 进行阻塞（Blocking）操作（如 IO 时）会阻塞掉整个程序。

yield 的使用

Python 对协程的支持是通过 generator 实现的。在 generator 中，不但可以通过 for 循环来迭代，还可以不断调用 next() 函数获取由 yield 语句返回的下一个值。

先把 yield 看做“return”，这个是直观的，它首先是个 return，普通的 return 是什么意思，就是在程序中返回某个值，返回之后程序就不再往下运行了。看做 return 之后再把它看做一个是生成器（generator）的一部分（带 yield 的函数才是真正的迭代器）。

【示例】yield 的使用

```
def foo():  
    print("starting...")  
    while True:  
        res = yield 4  
        print("res:",res)  
g = foo()  
print(next(g))  
print("*"*20)  
print(next(g))
```

执行结果：

```
starting...
4
*****
res: None
4
```

执行过程：

- 1.程序开始执行以后，因为 foo 函数中有 yield 关键字，所以 foo 函数并不会真的执行，而是先得到一个生成器 g(相当于一个对象)
- 2.直到调用 next 方法，foo 函数正式开始执行，先执行 foo 函数中的 print 方法，然后进入 while 循环
- 3.程序遇到 yield 关键字，然后把 yield 想成 return,return 了一个 4 之后，程序停止，并没有执行赋值给 res 操作，此时 next(g)语句执行完成，所以输出的前两行（第一个是 while 上面的 print 的结果,第二个是 return 出的结果）是执行 print(next(g))的结果，
- 4.程序执行 print("*"*20)，输出 20 个*
- 5.又开始执行下面的 print(next(g)),这个时候和上面那个差不多，不过不同的是，这个时候是从刚才那个 next 程序停止的地方开始执行的，也就是要执行 res 的赋值操作，这时候要注意，这个时候赋值操作的右边是没有值的（因为刚才那个是 return 出去了，并没有给赋值操作的左边传参数），所以这个时候 res 赋值是 None,所以接着下面的输出就是 res:None,
- 6.程序会继续在 while 里执行，又一次碰到 yield,这个时候同样 return 出 4，然后程序停止，print 函数输出的 4 就是这次 return 出的 4。

带 yield 的函数是一个生成器，而不是一个函数了，这个生成器有一个函数就是 next 函数，next 就相当于“下一步”生成哪个数，这一次的 next 开始的地方是接着上一次的 next 停止的地方执行的，所以调用 next 的时候，生成器并不会从 foo 函数的开始执行，只是接着上一步停止的地方开始，然后遇到 yield 后，return 出要生成的数，此步就结束。

【示例】yield 简单实现协程

```
import time
def A():
    while True:
        print('----A----')
        yield
        time.sleep(0.5)

def B(c):
    while True:
```

```
print('----B----')
c._next_()
time.sleep(0.5)

if __name__ == '__main__':
    a=A()
    B(a)
```

执行结果：

```
----B----
----A----
----B----
----A----
----B----
----A----
----B----
----省略----
```

send 发送数据

send 是发送一个参数给 res 的，因为上面讲到，return 的时候，并没有把 4 赋值给 res，下次执行的时候只好继续执行赋值操作，只好赋值为 None 了，而如果用 send 的话，开始执行的时候，先接着上一次（return 4 之后）执行，先把 10 赋值给了 res，然后执行 next 的作用，遇见下一回的 yield，return 出结果后结束。

【示例】yield 中 send 函数的使用

```
def foo():
    print("starting...")
    while True:
        res = yield 4
        print("res:",res)

g = foo()
print(next(g))
print("***20")
print(g.send(10))
```

执行结果：

```
starting...
```

4

res: 10

4

【示例】协程实现生产者消费者

```
import time
#生产者
def produce(c):
    c.send(None)
    for i in range(1,6):
        print('生产者生产%d 产品'%i)
        c.send(str(i))
        time.sleep(1)
#消费者
def customer():
    res=""
    while True:
        data = yield res
        if not data:
            return
        print('消费者消费%s 产品'%data)

if __name__ == '__main__':
    c=customer()
    produce(c)
```

执行结果：

```
生产者生产 1 产品
消费者消费 1 产品
生产者生产 2 产品
消费者消费 2 产品
生产者生产 3 产品
消费者消费 3 产品
生产者生产 4 产品
```

消费者消费 4 产品

生产者生产 5 产品

消费者消费 5 产品

异步 IO (asyncio) 协程

使用异步 IO，无非是提高我们写的软件系统的并发。这个软件系统，可以是网络爬虫，也可以是 Web 服务等等。

并发的方式有多种，多线程，多进程，异步 IO 等。多线程和多进程更多应用于 CPU 密集型的场景，比如科学计算的时间都耗费在 CPU 上，利用多核 CPU 来分担计算任务。多线程和多进程之间的场景切换和通讯代价很高，不适合 IO 密集型的场景。而异步 IO 就是非常适合 IO 密集型的场景，比如网络爬虫和 Web 服务。

IO 就是读写磁盘、读写网络的操作，这种读写速度比读写内存、CPU 缓存慢得多，前者的耗时是后者的成千上万倍甚至更多。这就导致，IO 密集型的场景 99% 以上的时间都花费在 IO 等待的时间上。异步 IO 就是把 CPU 从漫长的等待中解放出来的方法。

asyncio

asyncio 是 Python 3.4 版本引入的标准库，直接内置了对异步 IO 的支持。asyncio 的编程模型就是一个消息循环。我们从 asyncio 模块中直接获取一个 EventLoop 的引用，然后把需要执行的协程扔到 EventLoop 中执行，就实现了异步 IO。

(1) 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。

(2) 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。

(3) 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。

(4) 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

(5) 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：

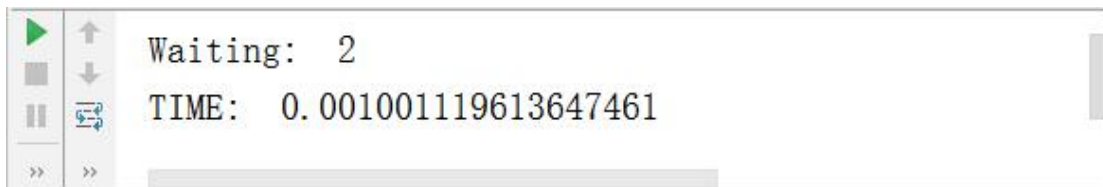
【示例】定义一个协程

```
import time, asyncio  
now = lambda : time.time()
```

#通过 async 定义一个协程，该协程不能直接运行，需要将协程加入到事件循环中

```
async def do_work(x):  
    print('waiting:%d'%x)  
  
start = now()  
#得到一个协程对象  
coroutine=do_work(2)  
#创建一个事件循环  
loop=asyncio.get_event_loop()  
#将协程对象加入到事件循环中  
loop.run_until_complete(coroutine)  
print('TIME: ', now() - start)
```

执行结果：



```
Waiting: 2  
TIME: 0.001001119613647461
```

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。

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。

`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 `future` 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`。

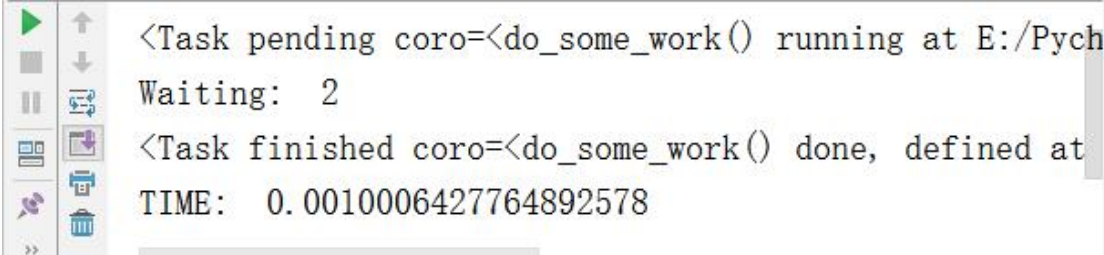
【示例】创建一个 task

```
import asyncio  
import time  
now = lambda: time.time()  
async def do_work(x):
```



```
print('Waiting: ', x)
start = now()
coroutine = do_work(2)
loop = asyncio.get_event_loop()
#创建一个 task
task = loop.create_task(coroutine)
#task=asyncio.ensure_future(coroutine)
print(task)
loop.run_until_complete(task)
print(task)
print("TIME: ", now() - start)
```

执行结果：



```
<Task pending coro=<do_some_work() running at E:/Pyth
Waiting: 2
<Task finished coro=<do_some_work() done, defined at
TIME: 0.0010006427764892578
```

创建 task 后，task 在加入事件循环之前是 pending 状态，因为 do_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。

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。

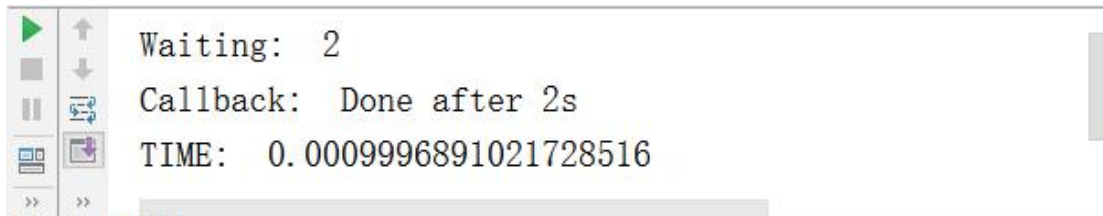
【示例】绑定回调

```
import time
import asyncio
now = lambda: time.time()
async def do_some_work(x):
    print('Waiting: ', x)
    return 'Done after {}'.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)
```

执行结果：



```
Waiting: 2
Callback: Done after 2s
TIME: 0.0009996891021728516
```

从上面实例可以看到，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。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，可以看到 task 有 finished 状态。在那个时候，可以直接读取 task 的 result 方法。

【示例】直接读取 task 的 result 方法

```
import time
import asyncio
now = lambda: time.time()
async def do_some_work(x):
    print('Waiting {}'.format(x))
    return 'Done after {}'.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))
```

执行结果：

```
Waiting 2
Task ret: Done after 2s
TIME: 0.0010006427764892578
```

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。

【示例】`asyncio.sleep` 函数来模拟 IO 操作

```
import asyncio
import time
now = lambda: time.time()
async def do_some_work(x):
    print('Waiting: ', x)
    await asyncio.sleep(x)
    return 'Done after {}'.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)
```

执行结果：

```
Waiting: 2
Task ret: Done after 2s
TIME: 2.0013554096221924
```

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。

【示例】asyncio 实现并发

```
import asyncio
import time

now = lambda: time.time()

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

执行结果：

```
Waiting: 1
Waiting: 2
Waiting: 4
Task ret: Done after 1s
Task ret: Done after 2s
Task ret: Done after 4s
TIME: 4.001962423324585
```

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 `asyncio` 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，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。

【示例】协程嵌套

```
import asyncio
import time

now = lambda: time.time()

async def do_some_work(x):
    print('Waiting: ', x)
    await asyncio.sleep(x)
    return 'Done after {}'.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)
    ]
    done, pending = await asyncio.wait(tasks)
    for task in done:
```

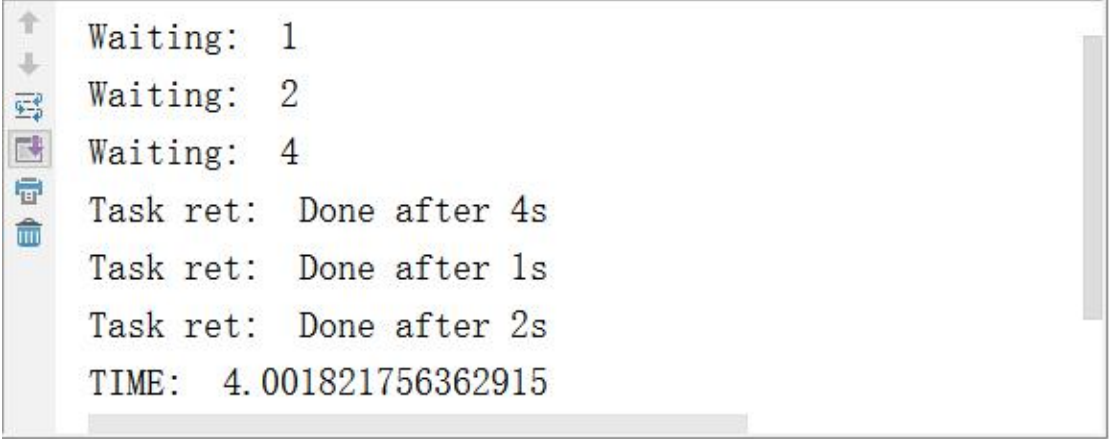
```
print('Task ret: ', task.result())

start = now()

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

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

执行结果：



```
Waiting: 1
Waiting: 2
Waiting: 4
Task ret: Done after 4s
Task ret: Done after 1s
Task ret: Done after 2s
TIME: 4.001821756362915
```

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。

【示例】`asyncio.gather` 创建协程对象

```
import asyncio
import time

now = lambda: time.time()

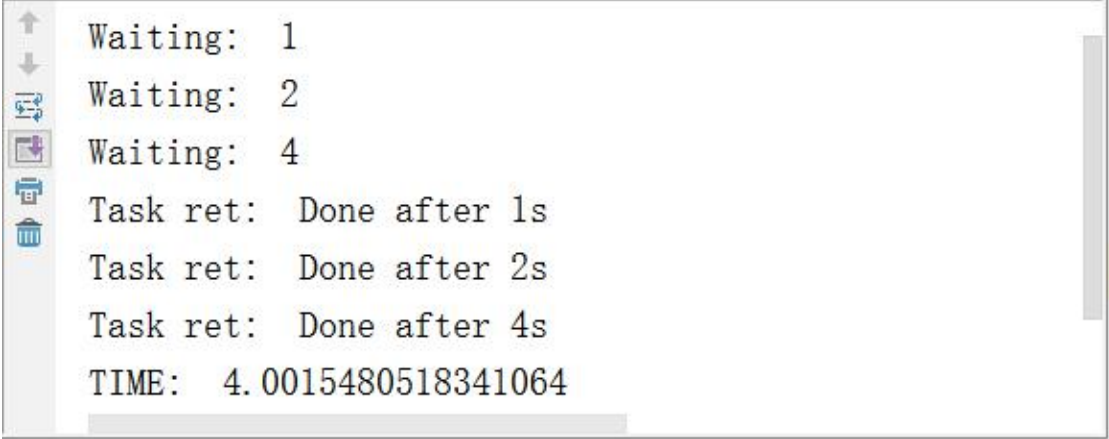
async def do_some_work(x):
    print('Waiting: ', x)
    await asyncio.sleep(x)
    return 'Done after {}'.format(x)

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

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

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

执行结果：



```
Waiting: 1
Waiting: 2
Waiting: 4
Task ret: Done after 1s
Task ret: Done after 2s
Task ret: Done after 4s
TIME: 4.0015480518341064
```

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。

【示例】不在 main 协程函数里处理结果

```
import asyncio
import time

now = lambda: time.time()

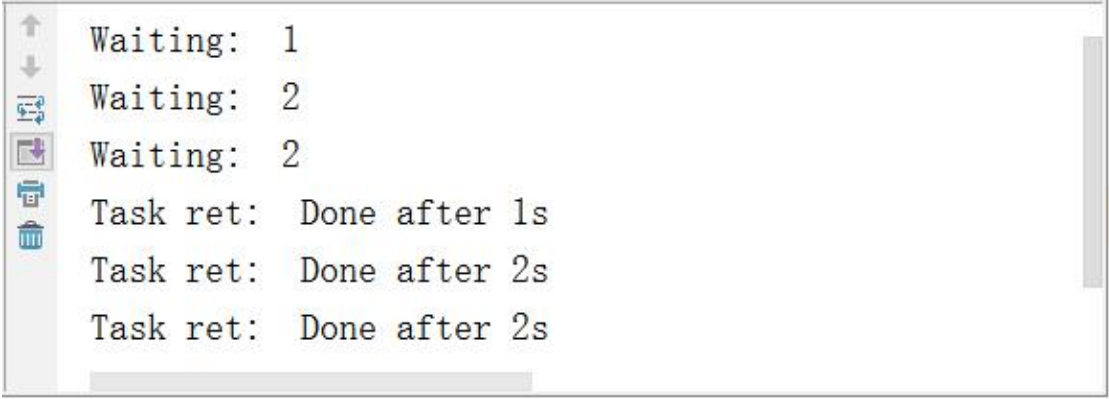
async def do_some_work(x):
    print('Waiting: ', x)
    await asyncio.sleep(x)
    return 'Done after {}'.format(x)

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)
```

```
]
    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)
```

执行结果：



```
Waiting: 1
Waiting: 2
Waiting: 2
Task ret: Done after 1s
Task ret: Done after 2s
Task ret: Done after 2s
```

或者返回使用 `asyncio.wait` 方式挂起协程。

【示例】使用 `asyncio.wait` 方式挂起协程

```
import asyncio
import time
now = lambda: time.time()

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



```

    return await asyncio.wait(tasks)

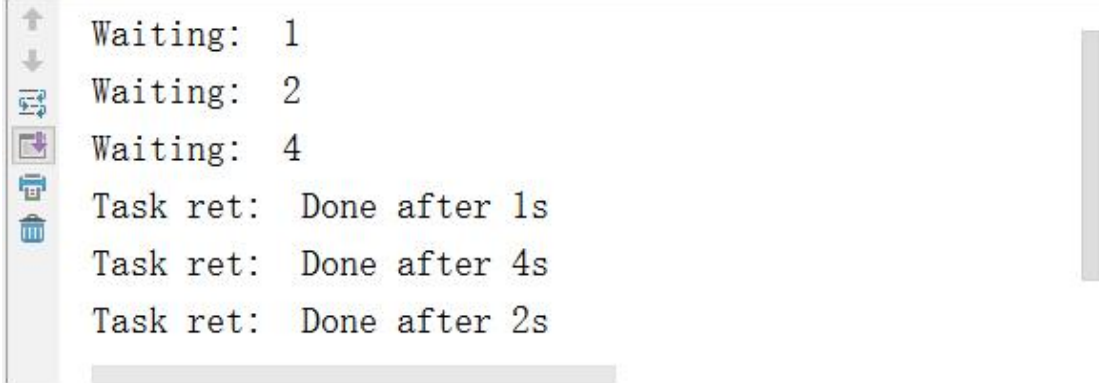
start = now()

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

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

```

执行结果：



```

Waiting: 1
Waiting: 2
Waiting: 4
Task ret: Done after 1s
Task ret: Done after 4s
Task ret: Done after 2s

```

【示例】使用 asyncio 的 as_completed 方法

```

import asyncio
import time

now = lambda: time.time()

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

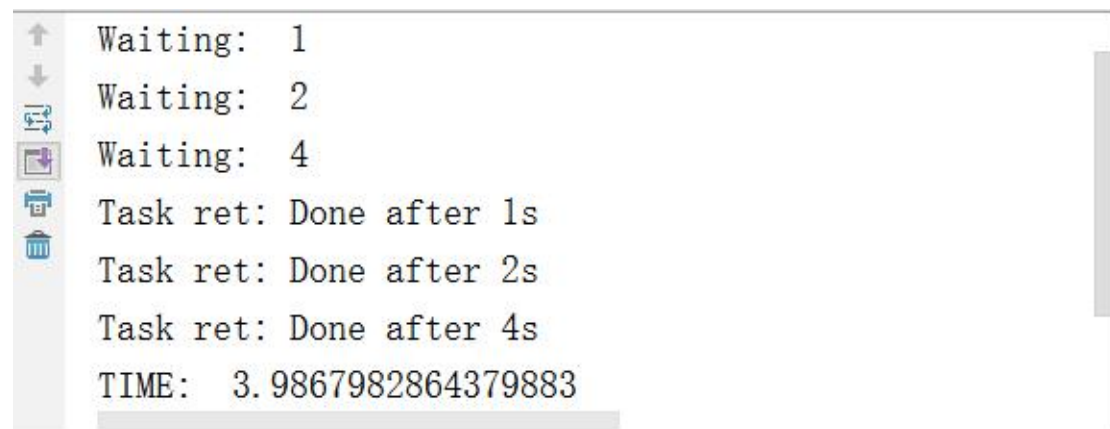
```

```
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)
```

执行结果：



```
Waiting: 1
Waiting: 2
Waiting: 4
Task ret: Done after 1s
Task ret: Done after 2s
Task ret: Done after 4s
TIME: 3.9867982864379883
```

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：

- (1) Pending
- (2) Running
- (3) Done
- (4) 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`。

【示例】协程停止

```
import asyncio
import time

now = lambda: time.time()

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

```
await asyncio.sleep(x)
return 'Done after {}'.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 `KeyboardInterrupt`。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 12停止协程.py:6> wait_for=<Future pending cb=[<TaskWakeupMethWrapper object at 0x0000014849B70A08>()]> cb=[_wait.<locals>._on_completion() at E:\soft\Python36\lib\asyncio\tasks.py:380]>, <Task pending coro=<do_some_work() running at 12停止协程.py:6> wait_for=<Future pending cb=[<TaskWakeupMethWrapper object at 0x0000014849BFED38>()]> cb=[_wait.<locals>._on_completion() at E:\soft\Python36\lib\asyncio\tasks.py:380]>], <Task pending coro=<do_some_work() running at 12停止协程.py:6> wait_for=<Future pending cb=[<TaskWakeupMethWrapper object at 0x0000014849BFED38>()]> cb=[_wait.<locals>._on_completion() at E:\soft\Python36\lib\asyncio\tasks.py:380]>], <Task pending coro=<do_some_work() running at 12停止协程.py:6> wait_for=<Future pending cb=[<TaskWakeupMethWrapper object at 0x0000014849BFED38>()]> cb=[_wait.<locals>._on_completion() at E:\soft\Python36\lib\asyncio\tasks.py:380]>], <Task pending coro=<wait() running at E:\soft\Python36\lib\asyncio\tasks.py:313> wait_for=<Future pending cb=[<TaskWakeupMethWrapper object at 0x0000014849BFE88>()]>>>}
    ]
True
True
True
True
```

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。

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 相当于最外的一个 task，那么处理包装的 main 函数即可。

【示例】把 task 的列表封装在 main 函数中，协程停止

```
import asyncio
import time

now = lambda: time.time()

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

```