协程

协程（Coroutine），又可以称作为微线程。协程是一种用户态轻量级的线程。

线程是系统级别，是由系统来统一调度的；而协程是程序级别的，由开发者根据自己的需要来调度。

同一线程下的一段代码<1>执行着执行着就可以中断， 然后跳去执行另一段代码，当再次回来执行代码块<1>的时候，接着从之前中断的地方开始执行。这个就是协程。

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，恢复之前保存的寄存器上下文和栈。 所以协程能保留上一次调用时的状态，当每次过程重新载入时，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。

协Python3中内置了异步IO。遇到IO密集型的业务时，总是很费时间，多线程加上协程，能很好的改善这个情况。在WEB应用中效果尤为明显。

下面我们来具体学习协程。

1.生成器函数-> 协程

在 Python2.2 中，第一次引入了生成器，生成器实现了一种惰性、多次取值的方法，此时还是通过 next 构造生成迭代链或 next 进行多次取值。

直到在Python2.5 中，yield 关键字被加入到语法中，这时生成器有了记忆功能，下一次从生成器中取值可以恢复到生成器上次 yield 执行的位置。

之前的生成器都是关于如何构造迭代器，在 Python2.5 中生成器还加入了 send 方法，与 yield 搭配使用。

我们发现，此时，生成器不仅仅可以 yield 暂停到一个状态，还可以往它停止的位置通过 send 方法传入一个值改变其状态。

1.1 生成器函数定义协程：

```python
# 代码1.1
#! -*- coding: utf-8 -*-
import inspect

# 协程使用生成器函数定义：定义体中有yield关键字。
def simple_coroutine():
    print('-> coroutine started')
    # yield 在表达式中使用；如果协程只需要从客户那里接收数据，yield关键字右边不需要加
    x = yield
    print('-> coroutine received:', x)

# 和创建生成器的方式一样，调用函数得到生成器对象。
my_coro = simple_coroutine()

# 首先要调用next()函数，因为生成器还没有启动，没有在yield语句处暂停，
# 所以开始无法发送数据(发送 None 可以达到相同的效果 my_coro.send(None))
next(my_coro)
-> coroutine started

# 调用这个方法后，协程定义体中的yield表达式会计算出42；现在协程会恢复，一直运行到下一个
my_coro.send(42)

-> coroutine received: 42

# 这里，控制权流动到协程定义体的尾部，导致生成器像往常一样抛出StopIteration异常
Traceback (most recent call last):
  File "/Users/gs/coroutine.py", line 18, in <module>
    my_coro.send(42)
StopIteration
```

send方法的参数会成为暂停yield表达式的值，所以，仅当协程处于暂停状态是才能调用send方法。

如果协程还未激活（GEN_CREATED 状态）要调用next(my*coro) 激活协程，也可以调用 my*coro.send(None)

协程状态： 协程有四个状态，可以使用inspect.getgeneratorstate()函数确定：

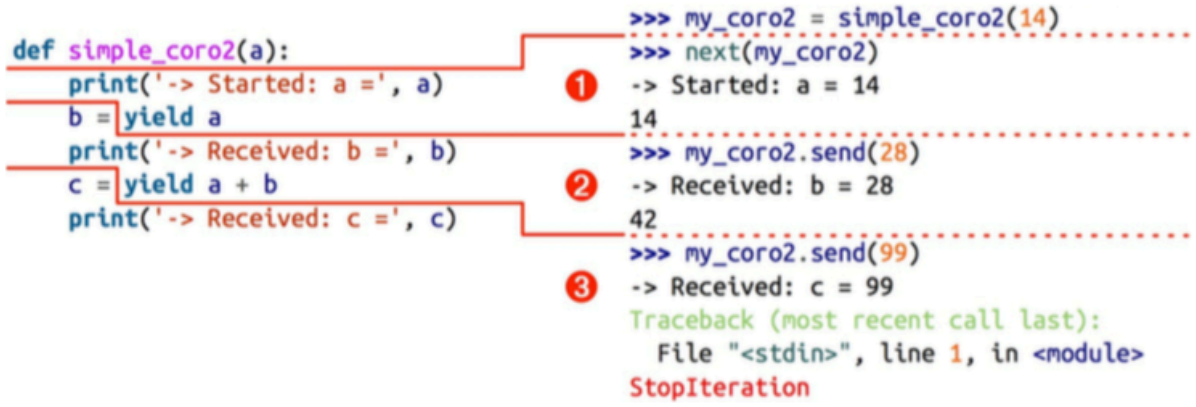
> - GEN_CREATED # 等待开始执行
> - GEN_RUNNING # 解释器正在执行（只有在多线程应用中才能看到这个状态）
> - GEN_SUSPENDED # 在yield表达式处暂停
> - GEN_CLOSED # 执行结束

产出两个值

```
# 代码1.2
def simple_coro2(a):
    print('-> Started: a=', a)
    b = yield a
    print('-> Received: b=', b)
    c = yield a + b
    print('-> Received: c=', c)

>>> my_coro2 = simple_coro2(14)
>>> next(my_coro2)
-> Started: a = 14
14
>>> my_coro2.send(28)
-> Received: b = 28
42
>>> my_coro2.send(99)
-> Received: c = 99
```

运行步骤解析：



思考：思考：有什么好的方式来预激活协程?

（装饰器）

2. 协程异常处理和终止

```python
# 代码1.3
#! -*- coding: utf-8 -*-

from functools import wraps

def coroutine(func):
    '''
    装饰器：  向前执行到第一个`yield`表达式，预激`func`
    :param func: func name
    :return: primer
    '''

    @wraps(func)
    def primer(*args, **kwargs):
        # 把装饰器生成器函数替换成这里的primer函数；调用primer函数时，返回预激后的生
        gen = func(*args, **kwargs)
        # 调用被被装饰函数，获取生成器对象
        next(gen)  # 预激生成器
        return gen  # 返回生成器
    return primer


@coroutine
def averager():
    # 使用协程求平均值
    total = 0.0
    count = 0
    average = None
    while True:
        term = yield average
        total += term
        count += 1
        average = total/count

coro_avg = averager()
print(coro_avg.send(40))
print(coro_avg.send(50))
print(coro_avg.send('123')) # 由于发送的不是数字，导致内部有异常抛出。
```

从Python2.5 开始，我们可以在生成器上调用两个方法，显式的把异常发给协程。 这两个方法是throw和close。

generator.throw(exc*type[, excvalue*[, traceback]]) 这个方法使生成器在暂停的yield表达式处抛出指定的异常。如果生成器处理了抛出的异常，代码会向前执行到下一个yield表达式，而产出的值会成为调用throw方法得到的返回值。 如果没有处理，则向上冒泡，直接抛出， 传到调用方的上下文中。

generator.close() 使得生成器在暂停的yield表达式处抛出GeneratorExit异常。如果生成器没有处理这个异常或者抛出StopIteration异常（一般指运行到结尾），调用方不会报错。 如果收到StopIteration异常， 生成器一定不能产出值，否则解释器会抛出RuntimeError异常。

3. 协程返回值

```python
# 代码1.4
#! -*- coding: utf-8 -*-

from collections import namedtuple

Result = namedtuple('Result', 'count average')

def averager():
    total = 0.0
    count = 0
    average = None
    while True:
        term = yield
        if term is None:
            break  # 为了返回值，协程必须正常终止；这里是退出条件
        total += term
        count += 1
        average = total/count
    # 返回一个namedtuple，包含count和average两个字段。在python3.3前，如果生成器返
    return Result(count, average)

coro_avg = averager()
next(coro_avg)
coro_avg.send(10)
coro_avg.send(20)

try:
    coro_avg.send(None)
except StopIteration as exc:
    result = exc.value
```

如果生成器处理了抛出的异常，代码会向前执行到下一个yield表达式，而产出的值会成为调用throw方法得到的返回值。

4. yield from

python3.3 引入yield from 结构的主要原因之一与把异常传入嵌套的协程有关，另一个原因是让协程更方便的返回值。

yield from 用于简化for循环中的yield表达式。

```python
# 代码1.5
def generator_word():
    for c in 'ABCDE':
        yield c
    for i in range(1, 3):
        yield i
list(generator_word())
['A', 'B', 'C', 'D', 'E', 1, 2]


def generator_word():
    yield from 'ABCDE':
    yield from range(1, 3):
list(generator_word())
['A', 'B', 'C', 'D', 'E', 1, 2]
```

> - 在函数外部不能使用yield from（yield也不行）。
> - yield from 是 Python3.3 后新加的语言结构。和其他语言的await关键字类似，
> - yield from x 表达式对x对象做的第一件事是，调用 iter(x)，获取迭代器。所以要求x是可迭代对象
> - 解释器不仅会捕获StopIteration异常，还会把value属性的值变成yield from 表达式的值。

yield from 的主要功能是打开双向通道，把最外层的调用方与最内层的子生成器连接起来，使两者可以直接发送和产出值，还可以直接传入异常，而不用在中间的协程添加异常处理的代码。

yield from 包含几个概念：

委派生成器

包含yield from 表达式的生成器函数

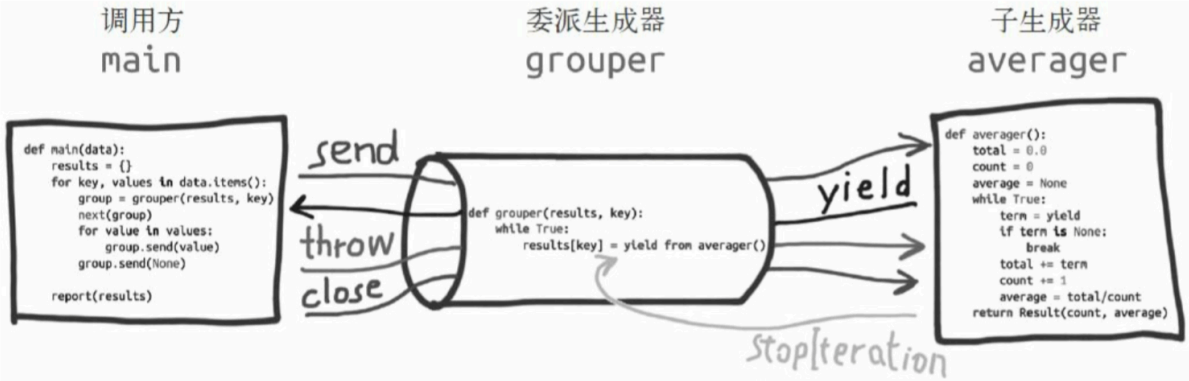
子生成器

从yield from 部分获取的生成器。

调用方

调用委派生成器的客户端（调用方）代码

下图很好的说明了yield from 的用法



上图：委派生成器在yeild from 表达式处暂停，调用方可以直接把数据发给子生成器，子生成器再把产出的值发给调用方。子生成器返回后，解释器会抛出 StopIteration异常，并把返回值附加在异常对象上，此时委派生成器恢复。

Python

```python
# 代码1.6

#! -*- coding: utf-8 -*-

from collections import namedtuple


Result = namedtuple('Result', 'count average')


# 子生成器
# 这个例子和上边示例中的 averager 协程一样，只不过这里是作为字生成器使用
def averager():
    total = 0.0
    count = 0
    average = None
    while True:
        # main 函数发送数据到这里
        term = yield
        if term is None: # 终止条件
            break
        total += term
        count += 1
        average = total/count
    return Result(count, average) # 返回的Result 会成为grouper函数中yield fro


# 委派生成器
def grouper(results, key):
     # 这个循环每次都会新建一个averager 实例，每个实例都是作为协程使用的生成器对象
    while True:
        # grouper 发送的每个值都会经由yield from 处理，通过管道传给averager 实例。
```

```
        # grouper会在yield from表达式处暂停，等待averager实例处理客户端发来的值。
        # averager实例运行完毕后，返回的值绑定到results[key] 上。while 循环会不断
        results[key] = yield from averager()


# 调用方
def main(data):
    results = {}
    for key, values in data.items():
        # group 是调用grouper函数得到的生成器对象，传给grouper函数的第一个参数是re
        # 用于收集结果；第二个是某个键
        group = grouper(results, key)
        next(group)
        for value in values:
            # 把各个value传给grouper 传入的值最终到达averager函数中；
            # grouper并不知道传入的是什么，同时grouper实例在yield from处暂停
            group.send(value)
        # 把None传入groupper，传入的值最终到达averager函数中，导致当前实例终止。然
        # 如果没有group.send(None)，那么averager子生成器永远不会终止，委派生成器也
        # 也就不会为result[key]赋值
        group.send(None)
    report(results)


# 输出报告
def report(results):
    for key, result in sorted(results.items()):
        group, unit = key.split(';')
        print('{:2} {:5} averaging {:.2f}{}'.format(result.count, group, r


data = {
    'girls;kg':[40, 41, 42, 43, 44, 54],
    'girls;m': [1.5, 1.6, 1.8, 1.5, 1.45, 1.6],
    'boys;kg':[50, 51, 62, 53, 54, 54],
    'boys;m': [1.6, 1.8, 1.8, 1.7, 1.55, 1.6],
}

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

> 这段代码从一个字典中读取男生和女生的身高和体重。然后把数据传给之前定义的 averager 协程，最后生成一个报告。

这段代码展示了yield from 结构最简单的用法，只有一个委派生成器和一个子生成器。委派生成器相当于管道，所以可以把任意数量的委派生成器连接在一起：一个委派生成器使用yield from 调用一个子生

成器，而那个子生成器本身也是委派生成器，使用yield from调用另一个生成器。最终以一个只是用yield表达式的生成器（或者任意可迭代对象）结束。整个联调都是以客户端来驱动。

5. 协程实现

5.1 asyncio + yield from

asyncio包详解见下一章

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

@asyncio.coroutine

```python
# 代码1.7
import asyncio

@asyncio.coroutine
def hello():
    print("Hello world!")
    # 异步调用asyncio.sleep(1):
    r = yield from asyncio.sleep(1)
    print("Hello again!")

# 获取EventLoop:
loop = asyncio.get_event_loop()
# 执行coroutine
loop.run_until_complete(hello())
loop.close()
```

5.2 asyncio.async(…)* await 函数

```python
async def hello():
    print("Hello world!")
    # 异步调用asyncio.sleep(1):
    r = await asyncio.sleep(1)
    print("Hello again!")

# 获取EventLoop:
loop = asyncio.get_event_loop()
# 执行coroutine
loop.run_until_complete(hello())
loop.close()
```

5.3 greenlet

可以实现协程，不过每一次都要人为的去指向下一个该执行的协程

5.4 gevent

比greenlet更强大的并且能够自动切换任务的模块gevent, gevent每次遇到io操作，需要耗时等待时，会自动跳到下一个协程继续执行。

使用猴子补丁，gevent能够修改标准库里面大部分的阻塞式系统调用，包括socket、ssl、threading和select等模块，而变为协作式运行。也就是通过猴子补丁的monkey.patch_xxx()来将python标准库中模块或函数改成gevent中的响应的具有协程的协作式对象。这样在不改变原有代码的情况下，将应用的阻塞式方法，变成协程式的。

Python

```python
# python2
from gevent import monkey; monkey.patch_all()
import gevent
import urllib2

def f(url):
    print('GET: %s' % url)
    resp = urllib2.urlopen(url)
    data = resp.read()
    print('%d bytes received from %s.' % (len(data), url))

gevent.joinall([
        gevent.spawn(f, 'https://www.python.org/'),
        gevent.spawn(f, 'https://www.yahoo.com/'),
        gevent.spawn(f, 'https://github.com/'),
])
```

6. 协程优缺点

优点

> - 协程轻量级的，不需要系统调度，故可以提高性能；
> - 没有原子操作，同步、锁
> - 高并发：一个CPU支持上万的协程都不是问题

缺点

> - 不能利用多核资源：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
> - 阻塞操作会阻塞调整个程序。

asyncio

asyncio是Python3.4版本引入的标准库，至少是Python3.3(需要手动安装）才可以使用。因为自 3.3 版本以来，

yield from 的引入使 Python 具备运行 asyncio 的基本条件。并发编程同时执行多条独立的逻辑流，即使它们是都工作在同个线程中的。每个协程都有独立的栈空间。

3.5版本中， 新语法的引入使Python具备了原生协程，而不再是一种新的生成器类型。

asyncio包使用事件循环驱动的协程实现并发。

4.1 Future

不用回调方法编写异步代码后，为了获取异步调用的结果，引入一个 Future 未来对象。Future 封装了与 loop 的交互行为，add*done*callback 方法向 epoll 注册回调函数，当 result 属性得到返回值后，会运行之前注册的回调函数，向上传递给 coroutine。

但是，每一个角色各有自己的职责，用 Future 向生成器 send result 以恢复工作状态并不合适，Future 对象本身的生存周期比较短，每一次注册回调、产生事件、触发回调过程后工作已经完成。所以这里又需要在生成器协程与 Future 对象中引入一个新的对象 Task，对生成器协程进行状态管理。

Future with run_until_complete() 执行事件循环，直到Future完成。返回Future结果或者对应的异常。

```python
import asyncio

async def slow_operation(future):
    await asyncio.sleep(1)
    future.set_result('Future is done!')

loop = asyncio.get_event_loop()
future = asyncio.Future()
asyncio.ensure_future(slow_operation(future))
loop.run_until_complete(future)
print(future.result())
loop.close()
```

Future with run_forever() 执行事件循环，直到调用stop()方法。

```python
import asyncio

async def slow_operation(future):
    await asyncio.sleep(1)
    future.set_result('Future is done!')

def got_result(future):
    print(future.result())
    loop.stop()

loop = asyncio.get_event_loop()
future = asyncio.Future()
asyncio.ensure_future(slow_operation(future))
future.add_done_callback(got_result)
try:
    loop.run_forever()
finally:
    loop.close()
```

PS:Python 里另一个 Future 对象是 concurrent.futures.Future，与 asyncio.Future 互不兼容，但容易产生混淆。

concurrent.futures 是线程级的 Future 对象，当使用 concurrent.futures.Executor 进行多线程编程时用于在不同的 thread 之间传递结果。

4.2 Task

Task，顾名思义，是维护生成器协程状态处理执行逻辑的的任务，Task 内的_step 方法负责生成器协程与 EventLoop 交互过程的状态迁移:向协程 send 一个值，恢复其工作状态，协程运行到断点后，得到新的未来对象，再处理 future 与 loop 的回调注册过程。

Task function

> Note In the functions below, the optional loop argument allows explicitly setting the event loop object used by the underlying task or coroutine. If it's not provided, the default event loop is used

ensure_future()

函数asyncio.ensure*future(coro*or_future, *, loop=None)

返回一个任务对象或Future对象。

wait()

wait(futures, *, loop=None, timeout=None, return*when=ALL*COMPLETED) 可以获取一个future的列表, 同时返回一个将它们全包括在内的单独的协程(done, pending)。类似concurrent.futures.wait()

所以我们可以这样写：

```python
loop.run_until_complete(asyncio.wait([print_page('http://example.com/foo')
                                      print_page('http://example.com/bar')
```

as_completed()

asyncio.as_completed(fs, *, loop=None, timeout=None)， 通过它可以获取一个协程的列表，同时返回一个按完成顺序生成协程的迭代器，因此当你用它迭代时，会尽快得到每个可用的结果。

```
return an iterator whose values are coroutines.

result = loop.run_until_complete(asyncio.as_completed(*tasks)

for f in as_completed(fs):
    result = yield from f  # The 'yield from' may raise
    # Use result
```

gather()

gather 的返回值为协程运行的结果。

```
Return a future aggregating results from the given coroutines
    or futures.

result = loop.run_until_complete(asyncio.gather(*tasks)
for result in results:
   print('Task ret: ', result)
```

4.3 Loop

事件循环的工作方式与用户设想存在一些偏差，理所当然的认知应是每个线程都可以有一个独立的 loop。但是在运行中，在主线程中才能通过 asyncio.get_event_loop() 创建一个新的 loop，而在其他线程时，使用 get_event_loop() 却会抛错，正确的做法应该是 asyncio.set_event_loop() 进行当前线程与 loop 的显式绑定。由于 loop 的运作行为并不受 Python 代码的控制，所以无法稳定的将协程拓展到多线程中运行。

协程在工作时，并不了解是哪个 loop 在对其调度，即使调用 asyncio.get_event_loop() 也不一定能获

取到真正运行的那个 loop。因此在各种库代码中，实例化对象时都必须显式的传递当前的 loop 以进行绑定。

4.4 async与await

Python3.5 中引入了这两个关键字用以取代 asyncio.coroutine 与 yield from，从语义上定义了原生协程关键字，避免了使用者对生成器协程与生成器的混淆。这个阶段(3.0-3.4)使用 Python 的人不多，因此历史包袱不重，可以进行一些较大的革新。

await 的行为类似 yield from，但是它们异步等待的对象并不一致，yield from 等待的是一个生成器对象，而await接收的是定义了__await__方法的 awaitable 对象。

在 Python 中，协程也是 awaitable 对象。

4.5 asynico 协程

- async def函数必定是协程，即使里面不含有await语句。
- 如果在async函数里面使用yield或yield from语句，会引发SyntaxError异常。
- 调用一个普通生成器，返回一个生成器对象（generator object）；相应的，调用一个协程返回一个协程对象（coroutine object）。
- 协程不再抛出StopIteration异常

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 我们需要一个事件循环。

我们可以通过asyncio.get*event*loop()得到一个标准的事件循环，之后使用它的run*until*complete()方法来运行协同程序。 所以，为了使之前的协同程序运行，我们只需要做下面的步骤：

```
loop = asyncio.get_event_loop()
loop.run_until_complete(print_page('http://example.com'))
```

> 除非想阻塞主线程，从而冻结事件循环或者整个应用，否则不要在asyncio协程中使用time.sleep(delay)。如果协程需要在一段时间内什么也不做， 应该使用yield from asyncio sleep(delay)或者await asyncio.sleep(delay)。

装饰器+yield from

@types.coroutine()

types模块添加了一个新函数coroutine(fn)，使用它，“生成器实现的协程”和“原生协程”之间可以进行互操作。

```python
@types.coroutine
def process_data(db):
    data = yield from read_data(db)
    ...
```

@asyncio.coroutine把一个generator标记为coroutine类型

async + await

新的await表达式用于获得协程执行的结果：

```python
async def read_data(db):
    data = await db.fetch('SELECT ...')
    ...
```

以CPython内部，await使用了yield from的实现，但加入了一个额外步骤——验证它的参数类型。await只接受awaitable对象。

如果在async def函数之外使用await语句，会引发SyntaxError异常。这和在def函数之外使用yield语句一样。

如果await右边不是一个awaitable对象，会引发TypeError异常。

```
import asyncio
import datetime

async def display_date(loop):
    end_time = loop.time() + 5.0
    while True:
        print(datetime.datetime.now())
        if (loop.time() + 1.0) >= end_time:
            break
        await asyncio.sleep(1)

loop = asyncio.get_event_loop()
# Blocking call which returns when the display_date() coroutine is done
loop.run_until_complete(display_date(loop))
loop.close()
```

协程嵌套

```python
import asyncio

async def compute(x, y):
    print("Compute %s + %s ..." % (x, y))
    await asyncio.sleep(1.0)
    return x + y

async def print_sum(x, y):
    result = await compute(x, y)
    print("%s + %s = %s" % (x, y, result))

loop = asyncio.get_event_loop()
loop.run_until_complete(print_sum(1, 2))
loop.close()
```

异步上下文管理器和“async with”（了解）

异步上下文管理器（asynchronous context manager），可以在它的enter和exit方法里挂起、调用异步代码。

为此，我们设计了一套方案，添加了两个新的魔术方法：__aenter__和 __aexit__，它们必须返回一个awaitable。

```python
class AsyncContextManager:
    async def __aenter__(self):
        await log('entering context')

    async def __aexit__(self, exc_type, exc, tb):
        await log('exiting context')
```

数据库异步事务

```python
async def commit(session, data):
    ...

    async with session.transaction():
        ...
        await session.update(data)
        ...
```

异步迭代器 async for（了解）

通过异步迭代器实现的一个新的迭代语法如下：

```
async for TARGET in ITER:
    BLOCK
else:
    BLOCK2
```

思考：协程使用 下载 邮件