# 爬虫中的异步任务设计

## 课程内容与系统环境介绍

### 课程介绍

**1. 进程、线程与协程**

**2. IO操作与IO模型**

**3. IO设计模式**

**4. Python异步库使用与爬虫异步实现**

### 系统环境介绍

centos7系统 + Docker 18.03

Python环境所用Dockerfile

# 注意：pip.conf文件为了解决，pip安装时可能出现的如certificate verify failed的错误

FROM python:3.6.5-stretch

COPY pip.conf /root/.config/pip/

RUN pip install ipython

RUN pip install requests &&\

pip install aiohttp &&\

pip install tornado &&\

pip install gevent &&\

pip install incremental &&\

pip install twisted[tls] &&\

pip install eventlet &&\

pip install dnspython &&\

pip install celery[redis]

## 1. 进程、线程与协程

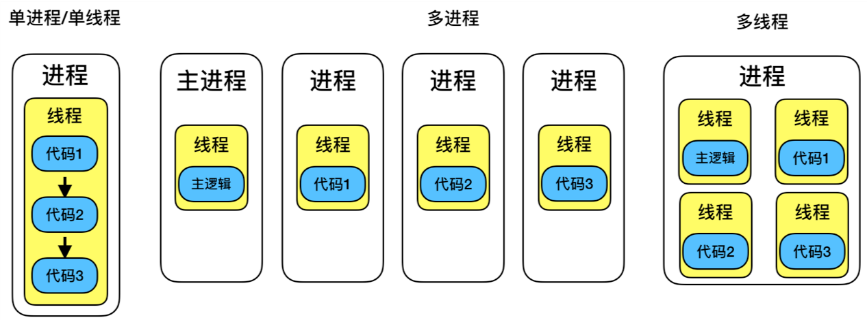
Python中实现程序的异步执行，都是更具进程、线程、协程实现

### 进程与线程

进程是具有一定独立功能的程序、它是系统进行资源分配和调度的一个独立单位，重点在系统调度和单独的单位，也就是说进程是可以独立运行的一段程序。

线程是进程的一个实体, 每个进程都至少有一个实体，一个线程又必须依赖一个进程, 线程是CPU调度和分派的基本单位，它是比进程更小的能独立运行的基本单位，资源是保存在进程中的, 线程自己基本上不拥有系统资源, 而是使用它所在的进程的资源。在运行时, 所需要的资源大都来自进程，只是暂用一些计数器、寄存器和栈。

1. 一个线程只能属于一个进程，而一个进程可以有多个线程，但至少有一个线程（通常说的主线程）。
2. 资源分配给进程，同一进程的所有线程共享该进程的所有资源。
3. 线程在执行过程中，需要协作同步。不同进程的线程间要利用消息通信的办法实现同步。
4. 处理机分给线程，即真正在处理机上运行的是线程。
5. 线程是指进程内的一个执行单元，也是进程内的可调度实体。



cpu在执行的时候, 都是以线程为单位进行执行和调度的, 多进程和多线程的本质都是多线程, 只不过多进程中的线程是处于不同的进程中的, 每个线程拥有自己单独的一份数据, 但需要手动来实现多进程数据共享的问题. 而多线程中的的多个线程是处在同一个进程中的, 它们共享同一个进程中的数据, 所以会出现线程的资源竞争和线程数据安全的问题.

### 协程

**思考：多进程、多线程、协程解决的问题？**

**答：解决多个任务的并行/并发执行**

* **并行：多个任务同时刻进行**。如服务员在点餐的同时、厨师在炒菜，同时刻做(非切换)多件事情. 多个人同时在做着多件事.
* **并发：多个任务交替执行。**如厨师一边炒菜、一边煲汤、一边烧肉，交替、切换着做多件事件。任务轮流交替着去执行, 只是一个人交替的在做多件事.

把人看成cpu, 做的事看成是函数, 并行相当于多个cpu在执行多个函数. 并发相当于一个cpu交替轮流的执行多个函数. 多进程本身就是并行的状态, 是多核cpu的每个核在执行一个函数. 而多线程并发则是在cpu一个核上轮流执行着多个任务.

程序中任务执行的本质是CPU作为执行者运行代码，因此并行必定意味着多核，并发好比单核切换交替执行多个任务。

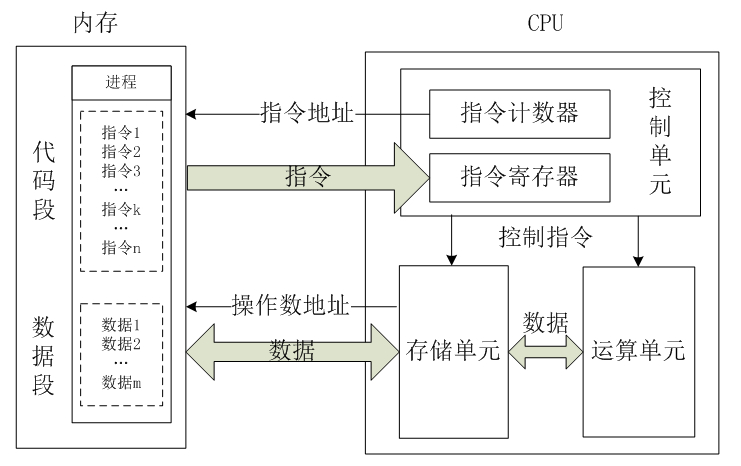
由于Python解释器中GIL全局解释器锁的问题，Python的多线程只能实现并发, 只能在cpu的一个核上运行，所以在python中, 多进程才能实现并行。而协程是在单线程中实现的，因此协程也只能实现并发。

进程和线程都是实际存在的对象，是操作系统中确确实实存在的东西；而协程又被称为微线程，是抽象出来的一个概念, 协程只是程序员虚拟出来的代码执行的流程。

多进程、多线程的切换是操作系统负责的，而协程的切换不是由操作系统的。而是由程序员进行控制的.

**协程的实现方式/原理**

协程就是为了解决并发/任务的交替执行的问题而出现的, 也就是说只要是单线程中实现了多个任务的交替/切换的执行，那么它就是协程。同时协程的切换本质上由用户控制。例如有三个任务, 使用一定的程序来控制它们在一个线程中交替轮流的执行, 就相当于实现了协程. 每一个任务完整的执行过程就是一个协程.



协程的实现方式其实都是通过将CPU的执行环境(控制单元、存储单元、运算单元)进行替换实现的。因为CPU只根据它控制单元、存储单元、运算单元里的东西来运算。

如当前CPU正在运行函数1，那么此时直接将函数1在CPU的运行情况保存起来，然后替换成函数2的，这样就实现了从函数1到函数2的切换。

具体实现一般分为两种情况：

* 一种是语言特性，原生支持，如Python的yield、Go的goroutine等；
* 另一种是第三方模块的单独实现，如gevent中的使用的greenlet。

**IO密集任务与计算密集任务**

* IO密集型：适合多线程或协程
  + 大量磁盘I/O操作（文件读写、磁盘读写）
  + 大量网络I/O操作（网络请求、socket程序）
* 计算密集型：适合多进程
  + 大量运算操作（计算精确到小数点1000位的圆周率、视频高清解码、图像运算）
  + 大量逻辑判断操作（循环判断、if判断等大量逻辑代码处理）
* 原因：
  + I/O操作不会占用CPU，多线程、协程能同时处理多个I/O操作. CPU在遇到I/O操作时, 默认情况下会挂起, 什么操作都不进行, 等待磁盘IO或网络IO数据的准备, 此时就会浪费大量的CPU资源. 使用多线程或协程来进行切换, 只要遇到I/O操作, 就切换到其它任务中进行操作. 对于大量的I/O操作, 多线程或协程的效率会远远大于多进程的效率. 大量IO操作时协程切换效率 > 线程 > 进程
  + 计算操作需要一直使用CPU，多进程能利用多个核心.
  + 多进程资源开销比多线程资源开销大很多. 所以除了计算密集型的任务之外, 一般都会优先使用多线程的模式来设计多任务.

**网络应用是I/O密集型应用**，爬虫程序往往是一个I/O密集型应用, 爬虫程序对应大量socket套接字的收发数据, 每一次收发数据的操作都是一次IO操作。

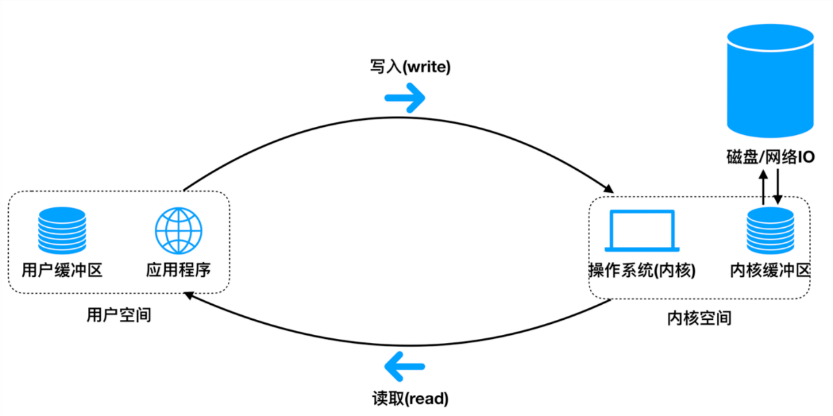
而显然针对I/O密集型程序，协程开销相较于线程小得多，因此后面将重点围绕协程相关的知识进行爬虫异步设计的学习。异步库也主要是针对网络I/O来实现异步的.

**注意：**如今大量程序开发都是网络I/O密集为主，如爬虫开发、web开发、数据库客户端开发等，真正的磁盘I/O密集主要是如mysql、postgresql、mongodb数据库服务端开发等，其内部的磁盘读写都是利用进程、线程来实现并发的。因此**后面所提及的I/O都是主要指网络I/O**。

## 2. IO操作与IO模型

### IO操作本质

数据复制的过程中为什么不会消耗CPU?



内存空间分为用户空间和内核空间, 用户空间为用户的第3方程序占用的内存空间, 内核空间为操作系统的系统级应用的运行所占用的内存空间. 用户空间中的数据和程序不能占用内核空间, 只能读取其中的数据, 必须有操作系统的授权才能写入数据. 操作系统所占用的内存和用户应用程序所占用的内存分别称为内核缓冲区和用户缓冲区, 操作系统和内核通过内核缓冲区与磁盘和网络进行IO操作, 应用程序通过用户缓冲区与操作系统/内核进行交互.

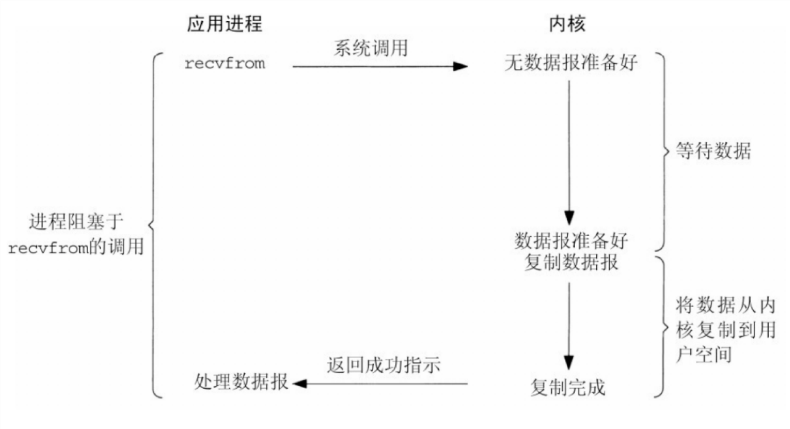
用户应用程序会发起IO操作, 但是真正的IO操作并不是应用程序直接完成的, 而是由操作系统的内核来完成的. 应用程序向操作系统进行IO请求, 最终由操作系统完成真正的IO操作. 当操作系统从磁盘或网络IO中获取到数据后, 数据是到达操作系统占用的内核缓冲区的, 用户应用程序要想使用IO数据, 必须要把数据从内核缓冲区读取到用户缓冲区中, 操作系统拿到数据后, 一般会通知用户空间中的应用程序, 用户程序就会去内核缓冲区中读取数据, 把数据复制到用户缓冲区中. 由于写入和读取的操作都是复制的操作, 是不占用cpu的, 所以整个过程不消耗CPU. 用户应用程序在等待操作系统进行IO操作的过程中都会挂起, 直到数据准备好. 所以要想加快程序整体的运行速度, 就可以在挂起的时间使用协程切换到其它任务中, 先去进行其它的操作, 等到数据准备好, 再切换到本任务中继续执行.

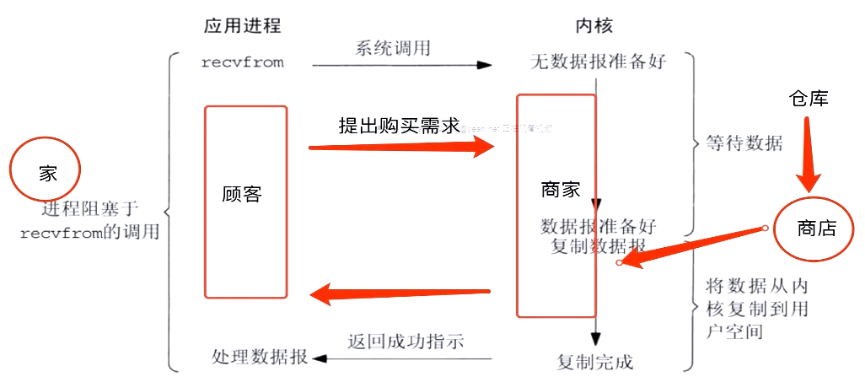
### IO模型

操作系统在完成I/O操作时的机制就称为IO模型.

**1. BIO – 阻塞模式I/O**

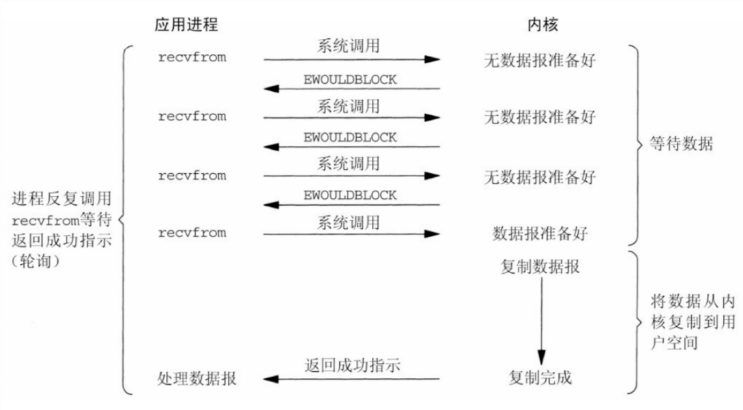
用户进程从发起请求，到最终拿到数据前，一直挂起等待； 数据会由用户进程完成拷贝. 只能同时处理一个任务, 一个任务处理完成后再处理下一个任务.





**2. NIO – 非阻塞模式I/O**

用户进程发起请求，如果数据没有准备好，那么立刻告知用户进程未准备好；此时用户进程可选择继续发起请求、或者先去做其他事情，稍后再回来继续发请求，直到被告知数据准备完毕，可以开始接收为止； 数据会由用户进程完成拷贝

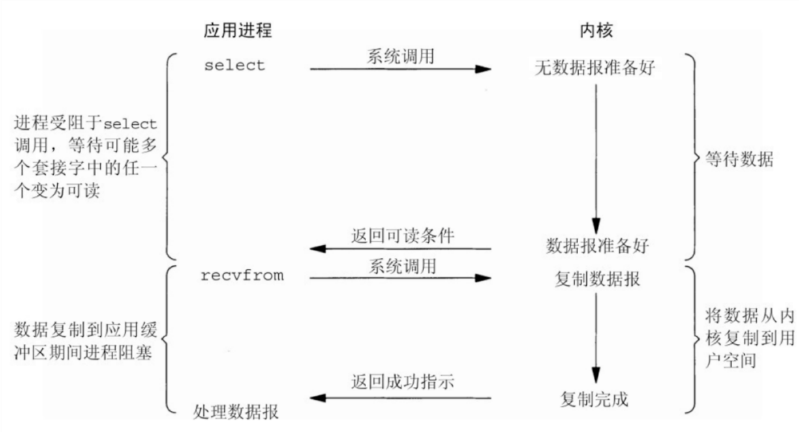


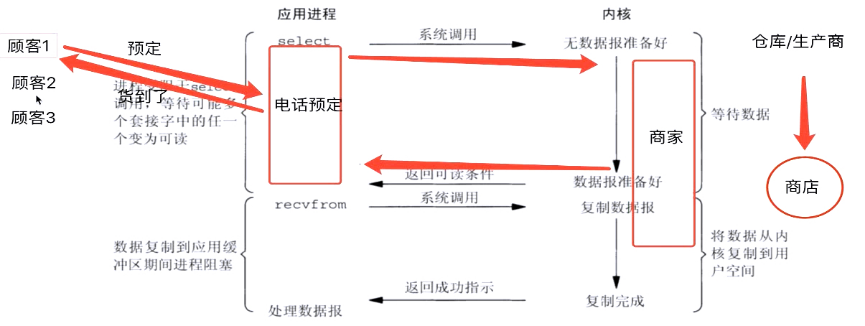
在等待内核准备数据的过程中是非阻塞的, 可以去做其它事情, 但从开始复制数据到用户接收完成的过程中是阻塞的, 不能去执行其它的操作. 在这个过程中, 其它的应用也都处于等待状态, 不能从内核中读取数据. 只有当一个程序数据接收结束时, 其它程序才继续执行. 非阻塞模式IO指的是在数据未准备好的过程是非阻塞的, 在复制数据的过程中是阻塞的.

用户购买商品的过程. 如果无货, 出去转转, 干些其它的. 一会儿回来问问, 直到有货.

**3. IO Multiplexing - I/O多路复用模型**

类似BIO，只不过找了一个代理，来挂起等待，并能同时监听多个请求； 数据会由用户进程完成拷贝





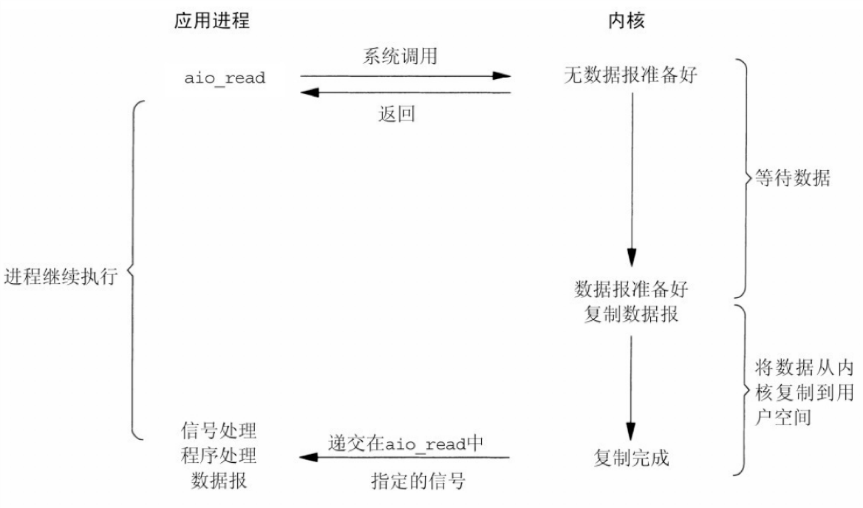
多个顾客通过商家提供的电话预定服务向商家预定不同的商品, 在某个商品有货时, 商家通知对应的顾客, 顾客再前往商家购买商品. 可以一次性的监听多个客户的请求. 在商品准备好时, 向对应的客户发送消息.

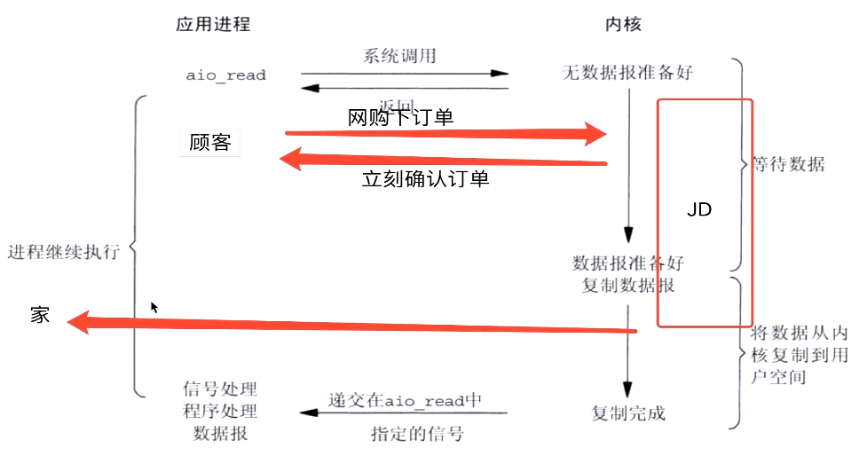
多个socket需要从硬盘/网络读取或向硬盘/网络写入数据, 就向select进行请求的注册, select相当于一个请求管理器, select开始监听各个socket之后, 向操作系统内核提交请求, 告诉操作系统哪些socket需要哪些数据. 操作系统逐一的去准备各个socket需要的数据, 一旦某个socket的数据准备好了, 就向对应的socket发送消息, 此socket就会从系统缓冲区中把数据复制到用户缓冲区中.

IO多路利用中, 整个程序还是单进程的程序, 但已经可以处理多个socket套接字的请求了. 但在select向操作系统内核发送数据请求的过程中是阻塞的, 在一个任务接收recvfrom数据的过程中, 程序也处于阻塞状态.

**4. AIO – 异步I/O模型**

发起请求立刻得到回复，不用挂起等待； 数据会由内核进程主动完成拷贝





多个顾客在家中网购, 向电商订购商品并提供地址, 电商立刻确认订单. 再由电商向用户发送包裹.

应用程序发送aio\_read的异步IO请求, 在请求中保存有用户缓冲区中保存数据的地址, 不管操作系统有没有准备好数据, 都会立即向应用程序返回一个信号, 操作系统内核就会开始准备数据, 一旦数据准备好, 系统就会自动把数据拷贝到用户请求中提供的地址中. 应用程序就可以直接从此地址中读取数据了. 应用进程没有在任何地方阻塞和等待, 直到缓冲区中出现需要的数据.

但AIO的效率不一定比IO多路复用的效率高.

任何过程中用户都没有进行等待. 但只有特定的操作系统才支持AIO模式. Windows支持, 如果是Windows Server, 可以使用这种模式的IO操作. Linux中的支持并不完善, 所以Linux中更常用的是IO多路复用.

### 同步I/O与异步I/O

判断一种IO模式是同步IO还是异步IO, 就是要在多任务的情况下, 判断其中一个任务会不会导致其它任务的阻塞.

* 同步I/O
  + 概念：导致请求进程阻塞的I/O操作，直到I/O操作任务完成
  + 类型：BIO、NIO、IO Multiplexing
* 异步I/O
  + 概念：不导致进程阻塞的I/O操作
  + 类型：AIO

注意：

* 同步I/O与异步I/O判断依据是，是否会导致用户进程阻塞
* BIO中socket直接阻塞等待（用户进程主动等待，并在拷贝时也等待）
* NIO中将数据从内核空间拷贝到用户空间时阻塞（用户进程主动询问，并在拷贝时等待）
* IO Multiplexing中select等函数为阻塞、拷贝数据时也阻塞（用户进程主动等待，并在拷贝时也等待）
* AIO中从始至终用户进程都没有阻塞（用户进程是被动的）

## 3. IO设计模式

通常采用事件驱动模型来实现高性能的I/O程序，一般分为两种模式：

- Reactor模式，基于同步I/O实现

- Proactor模式，基于异步I/O实现

Reactor模式通常采用IO多路复用机制进行具体实现

- kqueue、epoll、poll、select等机制

Proactor模式通常采用OS Asynchronous IO(AIO)的异步机制进行实现

- 前提是对应操作系统支持AIO，比如支持异步IO的linux(不太成熟)、具备IOCP的windows server(非常成熟)

Reactor模式和Proactor模式都是**事件驱动模式**，主要实现步骤：

1. **事件注册：**

将事件与事件处理器进行分离。将事件注册到事件循环中，将事件处理器单独管理起来，记录其与事件的对应关系。事件处理器就是事件触发成功或失败后要进行的处理. 如把购买火车票作为一个事件, 购买成功后进行什么样的处理, 购买失败后进行什么样的处理, 就是事件处理器.

1. **事件监听：**

启动事件循环，一旦事件已经就绪/完成，就立刻通知事件处理器. 这里的事件主要指IO事件, 监听IO事件的状态, 一旦IO事件被触发, 就会通知对应的事件处理器, 事件处理器再通知应用程序来取数据. 或者一旦应用程序复制数据完成, 就会通知事件处理器来进行下一步的处理.

1. **事件分发：**

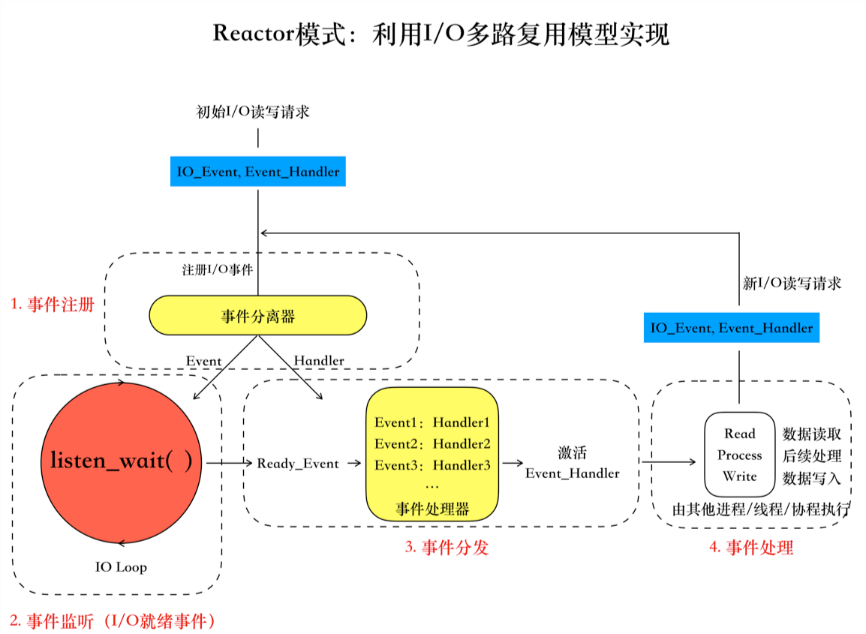
当收到事件就绪/完成的信号，便立刻激活与之对应的事件处理器

1. **事件处理：**

在进程/线程/协程中执行事件处理器. IO多路复用和AIO都只能实现在单个进程中监听多个套接字的读写操作, 但还没有实现对多个套接字异步处理的协调和控制, 还需要使用进程/线程/协程来实现多个套接字任务的切换.

事件驱动模型使用过程中，用户通常只负责**定义事件和事件处理器**并将其注册以及一开始的**事件循环的启动**，这个过程就会是以异步的形式执行任务。

### Reactor模式



1. 事件注册

注册事件和对应的事件处理器. 把事件和处理器进行分离, 但还可以通过处理器找到对应的事件.

2. 事件监听

本质上是一个事件/IO循环. 利用IO多路利用或AIO的机制来监听已经注册的套接字中哪些已经能够读取数据了或哪些套接字的数据已经读取完成了.

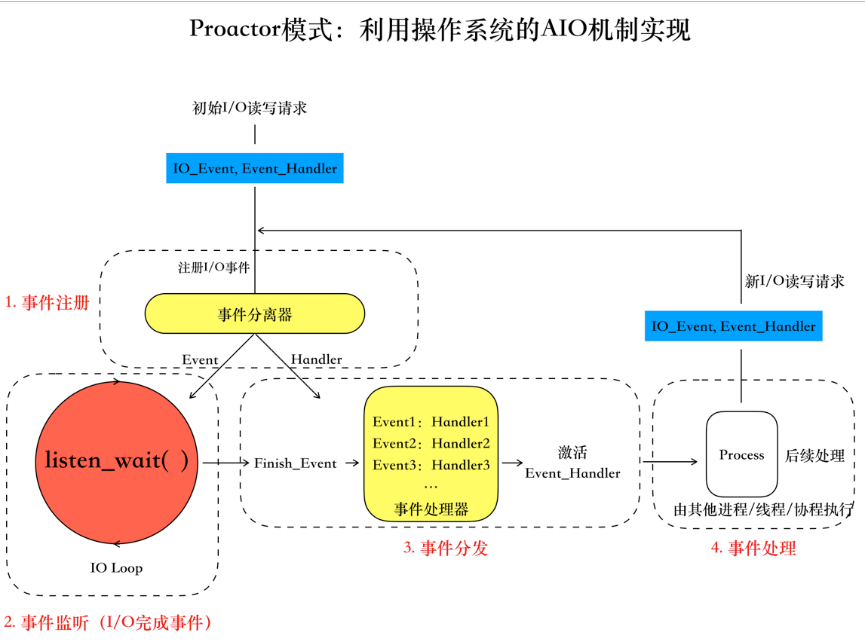
3. 事件分发

例如, 如果某个套接字可以读取数据了, 就把它从事件监听的IO Loop中弹出来, 成为Ready\_Event, 相当于这个事件已经处于准备就绪的状态, 就可以激活相应的事件处理器Event\_Handler.

4. 事件处理

激活的事件处理器Event\_Handler就可以使用进程/线程/协程进行异步的调用和执行. 此时就会把数据读取到内核空间中, 再进行处理.

### Proactor模式



数据的复制是由操作系统来完成的, Event\_Handler就不需要在事件处理的过程中进行read或write的操作了.

### 对比分析

Reactor模型基于同步IO实现, 处理耗时长的操作时会造成事件分发的阻塞，影响到后续事件的处理；在Reactor模式中, 如果在事件注册步骤中注册了多个套接字, 因为是基于同步IO的机制实现的, 如果事件处理器中的操作耗时较长, 在对IO Loop进行轮询时就会造成整个IO Loop执行效率的下降. 此时整个程序能够处理的并发量就会急剧下降. 因此特别是使用协程时, 事件处理器中的操作不要占用太多的时间.

Proactor模型实现逻辑复杂；依赖操作系统对异步的支持，目前实现了纯异步操作的操作系统少，实现优秀的如windows IOCP，但由于其windows系统用于服务器的局限性，目前应用范围较小；而Unix/Linux系统对纯异步的支持有限，因而应用事件驱动的主流还是基于select/epoll等实现的reactor模式

Python中：如asyncio、gevent、tornado、twisted等异步模块都是依据事件驱动模型设计，更多的都是使用reactor模型，其中部分也支持proactor模式，当然需要根据当前运行的操作系统环境来进行手动配置。

## 4. Python异步库使用与爬虫异步实现

Python中有多个基于协程的异步库，这些异步库都有一个共同特征：

* 对于IO事件，利用I/O多路复用或AIO等机制（即Reactor、Proactor），判断I/O操作是否就绪或完成，然后利用协程实现多个任务的切换，从而实现异步并发。都是利用协程实现的模块, 但也可以通过修改参数来把事件或事件处理器放到线程或进程池中运行.
* 对于非IO事件，直接利用协程实现多个任务的切换，从而实现异步并发。但以下4个模块并不都能实现非IO事件的协程自动切换, 如gevent就只能针对IO事件实现协程任务的自动切换.

异步库：都是利用协程实现的异步库.

* asyncio
* gevent
* tornado
* twisted

分布式异步任务框架

* celery

celery可以看成是以上4个异步库更上层的应用, 以上4个异步库相对于celery来说则是更底层的实现. celery默认情况下是使用线程池的模块来实现的异步, 但也可以使用gevent来实现异步.

基于celery实现爬虫的异步并发请求.

### 4.1. asyncio与爬虫

#### asyncio模块

[asyncio文档（python3.6）](https://www.rddoc.com/doc/Python/3.6.0/zh/library/asyncio/)

<https://www.rddoc.com/doc/Python/3.6.0/zh/library/asyncio/>

主要是学习如何基于事件驱动的思想实现使用asyncio实现单线程的并发代码.

asyncio原理：

* Reactor模型 + Proactor模型

asyncio内置对象：

* event loop事件循环对象：

程序开启一个无限循环，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即对应的事件处理器.

asyncio可以基于IO多路复用的Reactor模型来实现, 也可以使用基于AIO的Proactor模型来实现, 在Reactor模式中使用IO多路复用和在Proactor模式中使用AIO的主要目的是实现事件监听, 事件监听在整个过程中就是以IO Loop事件循环来实现的, 所以在asyncio中使用Reactor还是Proactor就是根据event\_loop的不同来进行区分的, 如果选择了基于epool的event\_loop, 就是Reactor模式的, 如果选择了基于AIO的, 就是使用的Proactor模型.

使用代码来实现. 查看asyncio文档 > 18.5.2. 事件循环 > class asyncio.SelectorEventLoop > class asyncio.ProactorEventLoop > SelectorEventLoop是基于selectors来实现的, 而selectors则有几个具体实现（KqueueSelector，EpollSelector ...）, 即SelectorEventLoop是基于IO多路复用来实现的. 而ProactorEventLoop类则只能在Windows上使用, 是基于AIO实现的.

* coroutine协程对象：

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。

例1：asyncio事件驱动代码中

# 1. 定义协程函数：使用async 关键词来定义函数, 调用函数时返回的就不是item, 而是一个返回一个函数整体上实现的async协程对象. 即coroutine协程对象. 这个协程对象就是事件循环中要处理的基本的对象. 如购买手机的事件, 在购买成功后就会调用callback中定义的事件处理器来进行进一步的处理.

async def buy(item):

return item

* Task对象：coroutine协程对象是事件循环中处理的基本的对象, 但实际中事件循环中处理的是Task对象, Task任务对象相当于对协程对象进行了进一步的封装，并设置了多种任务状态，便于管理. 事件循环中通过对Task对象的管理来实事件的调度. Task对象的状态包括以下几种:
  + Pending 挂起. 循环启动之前或者在执行其它task的过程中, 当前task对象处于pending状态.
  + Running 运行. 当前task正在被循环调用.
  + finished 完成
  + Cacelled 取消
* Future对象：与task对象基本一致, 使用方法也类似，在代码层面，Task是Future的子类, 继承自Future类, 在Future的基础上添加了上面的几种状态.

asyncio其他关键字：

* asyncio.corotines装饰器：在python3.5之前用于定义一个协程函数, 即是使用装饰器，内部使用yield from 挂起阻塞的异步调用接口；
* 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。

基于asyncio实现的单线程的并发的代码实现

##### 例1：asyncio事件驱动

'''asyncio事件驱动使用'''

import asyncio

import time

# 获取当前的时间戳, 以计算程序运行的时间

now = lambda: time.time()

# 事件处理器, 在代码中更多的称为是回调函数, 在task任务中调用的函数

# 事件处理器中的future参数本质上就是task对象. task对象是future对象的子类, 所以本质上就是task对象. 在执行loop.run\_until\_complete(task) 的过程中, 当task执行完成时, 会调用事件处理器, 执行event\_handler中的代码. 因为task是基于协程函数buy来创建的, task又继承自future, 这里的future.result() 就是在第2步中获取协程对象时传入的 "电脑" 参数.

def event\_handler(future):

if future:

print("购买'%s'成功, 开始玩游戏" % future.result())

else:

print("购买失败")

# 1. 定义协程函数：购买事件. 使用async 关键词来定义函数, 调用函数时返回的就不是item, 而是一个返回一个函数整体上实现的async协程对象. 即coroutine协程对象. 这个协程对象就是事件循环中要处理的基本的对象. 如购买手机的事件, 在购买成功后就会调用callback中定义的事件处理器 event\_handler 来进行进一步的处理.

async def buy(item):

return item

# 记录主程序开始运行的时间

start = now()

# 2. 调用协程函数获得协程对象

coroutine = buy("电脑")

# 3. 获取默认的事件循环对象, asyncio.get\_event\_loop() 是获取当前线程中的循环. 返回的是一个单例模式的对象, 即多次执行获得的对象是同一个对象.

loop = asyncio.get\_event\_loop()

# 4. 根据协程对象创建task对象: 注册事件event. task对象是对协程对象的封装. 也是事件循环对象真正去进行管理的对象. 可以使用三种不同的方法来创建task对象.

# 一旦执行了创建task对象的语句, 就把之前定义的buy事件注册到了当前的事件循环中.

# task = loop.create\_task(coroutine)

# task = asyncio.ensure\_future(coroutine)

task = asyncio.Task(coroutine)

# 5. 设置回调函数，也就是event\_handler. 当buy事件触发之后, 真正的去执行什么样的处理.

task.add\_done\_callback(event\_handler)

# 程序刚开始运行时, 事件Task会处于pending的状态. 也就是说事件在没有启动之前, 都是处于pending挂起的状态.

print(task)

# 6. 启动事件循环, 一旦启动了事件循环, 事件就会被监听, 被调用. 在本行代码执行的过程中, 会调用Task, 等待Task任务的运行, task会去执行协程函数buy中的代码, 返回一个协程对象, 因为在第5步中已经添加了回调函数即事件处理器, 所以只要Task一执行完毕, 就会调用事件处理器, 执行事件处理器中的代码. 所以这里会打印出事件处理器中的 "购买'电脑'成功, 开始玩游戏" 的提示.

loop.run\_until\_complete(task)

# 再打印task时, 它的状态就是finished的状态, 也就是执行完成的状态.

print(task)

print("耗时:", now()-start)

##### 例2：asyncio注册多个事件异步1

'''asyncio多个事件异步执行1'''

import asyncio

import time

now = lambda: time.time()

def event\_handler(future):

if future:

print("购买'%s'成功, 开始玩游戏"%future.result())

else:

print("购买失败")

# 1. 定义协程函数：购买手机的事件

async def buy(item):

# 利用asyncio.sleep()来暂停1s, 如果是同步状态的, 3个task执行的时间要大于3s, 但是程序最终运行的时间只有1s秒钟多一点, 所以这3个task是异步执行的. 也就执行到这里时, 就动态切换到其它任务去执行.

# 在使用yield把函数变成生成器时, 执行到yield时会把yield后面的内容返回, 同时交出代码的控制权, 去继续执行其它的代码. 这里的await的作用类似于yield from, await后面所跟的对象必须是task对象, 或协程对象coroutine对象, 或future对象. 代码执行到这里时, 会把await后面的对象返回, 同时交出代码的控制权, 继续去执行其它的代码. 所以整体上来看, 程序是单线程异步执行的, 程序整体的执行时间也是1s多一点.

await asyncio.sleep(1)

return item

start = now()

# 2. 获取默认的事件循环对象

loop = asyncio.get\_event\_loop()

tasks = []

# 创建并在事件循环中注册多个task对象.

for i in ["电脑", "手机", "掌机"]:

# 3. 根据协程对象创建task对象: 注册事件event

task = asyncio.ensure\_future(buy(i))

# 4.设置回调函数: 设置该事件的event\_handler

task.add\_done\_callback(event\_handler)

tasks.append(task)

# 5. 启动事件循环. run\_until\_complete中只能接受3种类型的对象. 一是协程对象, 二是task对象, 三是future对象. 因为tasks是一个列表的形式, 所以这里不能直接传入tasks, 而是使用asyncio.wait把用列表保存的3个task对象转换为future对象, 相当于把3个task对象同时注册进去. 现在, 3个任务就是并发执行的了.

# 此处打印的顺序就是for循环中定义task的顺序. 这是因为循环次数少, 创建的task对象数量少的原因.

loop.run\_until\_complete(asyncio.wait(tasks))

print("耗时:", now()-start)

##### 例3：asyncio注册多个事件异步2

注意：await后面必须是一个协程/future/task对象，asyncio.sleep返回的正是一个协程对象.

增加多个task, 再次执行代码, 可以看到task执行的顺序就不是创建task对象的顺序了. 这就是因为代码是异步执行的原因.

但是例1 - 例3中的事件都是非IO事件, 只是一个协程函数.

'''asyncio多个事件异步执行2'''

import asyncio

import time

import random

now = lambda: time.time()

def event\_handler(future):

if future:

print("购买'%s'成功, 开始玩游戏"%future.result())

else:

print("购买失败")

# 1. 定义协程函数：购买手机的事件

async def buy(item):

await asyncio.sleep(random.randint(0,10)\*0.1)

return item

start = now()

# 2. 获取默认的事件循环对象

loop = asyncio.get\_event\_loop()

tasks = []

for i in ["电脑", "手机", "掌机", "a1", 'a2', "a3"]:

# 3. 根据协程对象创建task对象: 注册事件event

task = asyncio.ensure\_future(buy(i))

# 4.设置回调函数: 设置该事件的event\_handler

task.add\_done\_callback(event\_handler)

tasks.append(task)

# 5. 启动事件循环

loop.run\_until\_complete(asyncio.wait(tasks))

print("耗时:", now()-start)

例1到例3中的代码都是以单线程的方式执行的, event\_handler和buy都是在当前进程的主线程中执行的, 可以使用threading模块来查看代码运行的进程.

*'''asyncio多个事件异步执行2'''*

**import** **asyncio**

**import** **time**

**import** **random**

**import** **threading**

now = **lambda**: time.time()

**def** event\_handler(future):

*# 打印执行此代码的线程的名称*

**print**("event\_handler: ", threading.current\_thread().name)

# 所有的代码都是在同一个线程即MainThread中执行的, 如果主线程中出现了堵塞, 所有代码必须要等待阻塞的代码执行完成才能进一步向下执行

# time.sleep(5)

**if** future:

**print**("购买'**%s**'成功, 开始玩游戏"%future.result())

**else**:

**print**("购买失败")

*# 1. 定义协程函数：购买手机的事件*

async **def** buy(item):

**print**("buy: ", threading.current\_thread().name)

await asyncio.sleep(random.randint(0,10)\*0.1)

**return** item

start = now()

*# 2. 获取默认的事件循环对象*

loop = asyncio.get\_event\_loop()

tasks = []

**for** i **in** ["电脑", "手机", "掌机", "a1", 'a2', "a3"]:

*# 3. 根据协程对象创建task对象: 注册事件event*

task = asyncio.ensure\_future(buy(i))

*# 4.设置回调函数: 设置该事件的event\_handler*

task.add\_done\_callback(event\_handler)

tasks.append(task)

*# 5. 启动事件循环*

loop.run\_until\_complete(asyncio.wait(tasks))

**print**("耗时:", now()-start)

此时再次执行代码, 可以看到, 6个buy和1个event\_handler都是在MainThread中执行的, 也就是都是在同一个线程中执行的, 如果在某一个执行过程中当前的主线程被阻塞了, 如在event\_handler中添加一个time.sleep(5), 所有的代码都会阻塞到这里, 直到阻塞的代码执行完成. 每执行一次task都会去调用一次event\_handler, 每次都会暂停5s, 所以在回调函数/事件处理器中尽量不要有耗时的操作, 一旦有耗时的操作, 如IO操作, 就要把它再次注册到事件循环IO Loop中. 而其它的耗时操作, 也尽量进行拆分处理或放到其它的线程或进程中进行处理. 如例4中, 在线程池中执行事件.

##### 例4：将事件放入线程池中执行

利用线程池执行task,

'''将事件放入线程池中执行'''

import asyncio

import time

import threading

now = lambda: time.time()

def event\_handler(future):

print("event\_handler thread: ", threading.current\_thread().name)

if future:

print("购买'%s'成功, 开始玩游戏"%future.result())

else:

print("购买失败")

# 事件函数,把事件函数放在线程池中运行. 注意此时并没有使用async来定义协程函数. 也就是说传递到进程/线程池中的函数只能是普通的函数, 不能是协程函数.

def buy(item):

print("buy thread: ", threading.current\_thread().name)

return item

start = now()

# 获取默认的事件循环对象

loop = asyncio.get\_event\_loop()

# 使用run\_in\_executor执行器：注册事件event. 这个执行器包含一个进程池/线程池.

# 使用loop.run\_in\_executor把buy事件放在线程池或进程池中执行, 第一个参数用来指定是使用线程池还是进程池, 如果使用None, 则表示使用默认的线程池, 即把buy放在一个线程池中运行. 如果想要使用进程池, 就先要创建出一个进程池的executor执行器替换到其中的None参数. run\_in\_executor的第二个参数是事件函数的函数名, 不是一个协程函数.

# loop.run\_in\_executor得到的是一个future对象.

future = loop.run\_in\_executor(None, buy, "电脑")

# 给future对象设置回调函数，也就是event\_handler

# 但回调还是主线程中运行

future.add\_done\_callback(event\_handler)

# 启动事件循环

loop.run\_until\_complete(future)

print("耗时:", now()-start)

运行程序, 事件函数buy是在ThreadPoolExecutor-0\_0中执行的, 而event\_handler 则是在MainThread中执行的. 也就是说即使事件是放在一个线程池中运行的, 事件处理器依然是在主线程中执行的. 所以回调函数中一定不要有太多的耗时操作, 否则主线程中会有长时间的阻塞, 降低整个程序执行的效率.

什么情况下需要把事件放在线程池中运行呢?

把事件放在线程中执行时, 事件函数必须是一个普通的函数, 不能是一个使用async定义的协程函数. 很多第3方模块在使用的过程中, 并没有考虑到与async模块的使用和兼容, 定义的很多函数都是普通函数, 如requests模块, 就可以使用loop.run\_in\_executor, 把requests中的操作放在线程池中执行. 例如:

import requests

future = loop.run\_in\_executor(None, requests.get, <http://www.baidu.com/>)

requests.get得到的是一个response对象, 使用future来接收, 这个response对象又会传递到主线程中交由event\_handler函数来执行.

即当使用的模块与async模块兼容性不好的情况下, 可以使用这种方式把函数放在线程池中执行.

在async模块的文档中搜索 run\_in\_executor,

coroutine AbstractEventLoop.run\_in\_executor(executor, func, \*args)

executor 参数应为 [**Executor**](https://www.rddoc.com/doc/Python/3.6.0/zh/library/concurrent.futures/#concurrent.futures.Executor) 实例。如果 executor 是 None，则使用默认执行器。

*在 3.5.3 版更改:***BaseEventLoop.run\_in\_executor()** 不再配置它创建的线程池执行器的 max\_workers，而是让它由线程池执行器（[**ThreadPoolExecutor**](https://www.rddoc.com/doc/Python/3.6.0/zh/library/concurrent.futures/#concurrent.futures.ThreadPoolExecutor)）设置为默认值。

查看[**Executor**](https://www.rddoc.com/doc/Python/3.6.0/zh/library/concurrent.futures/#concurrent.futures.Executor) 实例的说明. class concurrent.futures.Executor

17.4.2. ThreadPoolExecutor

[**ThreadPoolExecutor**](https://www.rddoc.com/doc/Python/3.6.0/zh/library/concurrent.futures/#concurrent.futures.ThreadPoolExecutor) 是一个 [**Executor**](https://www.rddoc.com/doc/Python/3.6.0/zh/library/concurrent.futures/#concurrent.futures.Executor) 子类，它使用一个线程池来异步执行调用。

17.4.3. ProcessPoolExecutor

[**ProcessPoolExecutor**](https://www.rddoc.com/doc/Python/3.6.0/zh/library/concurrent.futures/#concurrent.futures.ProcessPoolExecutor) 类是一个 [**Executor**](https://www.rddoc.com/doc/Python/3.6.0/zh/library/concurrent.futures/#concurrent.futures.Executor) 子类，它使用进程池来异步执行调用。 [**ProcessPoolExecutor**](https://www.rddoc.com/doc/Python/3.6.0/zh/library/concurrent.futures/#concurrent.futures.ProcessPoolExecutor) 使用 [**multiprocessing**](https://www.rddoc.com/doc/Python/3.6.0/zh/library/multiprocessing/#module-multiprocessing) 模块，它允许它侧面 [Global Interpreter Lock](https://www.rddoc.com/doc/Python/3.6.0/zh/glossary/#term-global-interpreter-lock)，但也意味着只有可拾取对象可以被执行和返回。

可以手动创建一个进程池对象, 作为run\_in\_executor的第一个参数, 使用进程池来运行回调函数.

##### 例5：协程链式嵌套（同步模式）

'''协程嵌套之链式嵌套'''

import asyncio

import time

now = lambda: time.time()

# 1. 定义协程函数. 除了task0之外, 每一个协程函数中都使用await调用了上一个协程函数.

async def task0():

await asyncio.sleep(1)

print("task0 done")

async def task1():

await task0()

await asyncio.sleep(1)

print("task1 done")

async def task2():

await task1()

await asyncio.sleep(1)

print("task2 done")

async def task3():

await task2()

await asyncio.sleep(1)

print("task3 done")

start = now()

# 获取默认的事件循环对象

loop = asyncio.get\_event\_loop()

# 获取task对象. 执行的时候是执行的task3(), task3中又调用task2, task2中调用task1, 逐次调用上一个协程函数.

task = asyncio.Task(task3())

# 启动事件循环

loop.run\_until\_complete(task)

print("耗时:", now()-start)

执行的时候是执行的task3(), task3中又调用task2, task2中调用task1, 逐次调用上一个协程函数. 所以最先完成的是task0, 然后是task1, task2, task3. 但最终耗时4s多, 也就是说, 以上4个协程函数的执行不同异步的了, 而是同步的了. 如果代码中的多个协程函数, 想要让这些函数以一定顺序的同步方式执行, 可以使用链式嵌套来完成. 也可以在task3中调用3个协程函数. 执行的顺序与上面的代码相同. 代码如下:

*# 1. 定义协程函数*

async **def** task0():

await asyncio.sleep(1)

**print**("task0 done")

async **def** task1():

await asyncio.sleep(1)

**print**("task1 done")

async **def** task2():

await asyncio.sleep(1)

**print**("task2 done")

async **def** task3():

await task0()

await task1()

await task2()

await asyncio.sleep(1)

**print**("task3 done")

##### 例6：协程组合嵌套（异步模式）

多个协程函数之间以异步的方式执行.

'''协程嵌套之组合嵌套'''

import asyncio

import time

now = lambda: time.time()

# 1. 定义协程函数

async def task0():

await asyncio.sleep(1)

print("task0 done")

return "task0"

async def task1():

await asyncio.sleep(1)

print("task1 done")

return "task1"

async def task2():

await asyncio.sleep(1)

print("task2 done")

return "task2"

async def task3():

await asyncio.sleep(1)

print("task3 done")

return "task3"

# 定义一个main函数.

async def main():

# 定义4个协程对象, 放在列表中.

tasks = [task0(), task1(), task2(), task3()]

# 0. 如果直接遍历tasks列表, 此时与协程函数的嵌套调用类似, 这里是同步的运行.

# for task in tasks:

# result = await task

# print(result)

# 有三种方式进行协程组合嵌套

# 1. asyncio.wait. 返回2个结果, dones为对应的task的返回值, pendings为对应的task的状态.

# dones, pendings = await asyncio.wait(tasks)

# for task in dones:

# print("Task result:", task.result())

'''运行结果: 异步运行、异步输出

task3 done

task1 done

task0 done

task2 done

Task result: task2

Task result: task0

Task result: task3

Task result: task1

耗时: 1.0027110576629639

'''

# 2. asyncio.as\_completed 返回的是一个可迭代对象

# for task in asyncio.as\_completed(tasks):

# result = await task

# print(result)

'''运行结果: 异步运行、异步输出, 结果顺序与调用顺序一致. 因为是逐个await进行task的调用的, 所以打印的结果与最终输出的结果的顺序相同, 也就是result和task是一一对应的关系. 因为结果是有顺序的, 所以可以对结果进行进一步的处理.

task2 done

task1 done

task3 done

task0 done

task2

task1

task3

task0

耗时: 1.0028691291809082

'''

# 3. asyncio.gather

results = await asyncio.gather(\*tasks)

print(results)

'''运行结果: 异步运行、同步输出, 结果顺序与传参顺序一致.

task0 done

task1 done

task2 done

task3 done

['task0', 'task1', 'task2', 'task3']

耗时: 1.0031750202178955

'''

start = now()

# 获取默认的事件循环对象

loop = asyncio.get\_event\_loop()

# 获取task对象

main\_task = asyncio.Task(main())

# 启动事件循环

loop.run\_until\_complete(main\_task)

print("耗时:", now()-start)

##### 例7：任务取消

import asyncio

import time

now = lambda: time.time()

# 1. 定义协程函数

async def task0():

print("task0 start")

await asyncio.sleep(5)

print("task0 done")

# 接收一个task参数, 1s后把task取消掉.

async def cancel\_task(task):

await asyncio.sleep(1)

print("取消前: ", task)

task.cancel()

print("取消后: ", task)

start = now()

# 获取默认的事件循环对象

loop = asyncio.get\_event\_loop()

# 获取task对象

main\_task = asyncio.Task(task0())

# 启动事件循环, 注册2个协程函数.

# 1秒后，main\_task将被取消

loop.run\_until\_complete(asyncio.wait([main\_task, cancel\_task(main\_task)]))

print("耗时:", now()-start)

"task0 start"

"取消前: " pending

"取消后: " cancelled

执行代码, 在取消前, task0是处于pending状态的. 为什么是task0处于挂起状态呢, 当运行到print("取消前:", task) 时, 主线程正在执行cancel\_task中的语名, 因为是单线程, 所以task0是处于挂起状态的. 取消task0的运行之后, task0是处于cancell状态的. 整个程序运行时间为1s多一点. 如果注释掉task\_cancel()时, 整个程序的运行时间为5s.

##### 例8：退出协程循环

显示退出循环. 之前task任务的运行都是调用oop.run\_until\_complete 来完成的, 也就是说注册的多个task的运行结束后才退出协程的运行, 还可以使用loop.run\_forever() 来启动循环, 让循环永久性的执行. 在程序运行的过程中需要动态的添加一些task, 如果以loop.run\_until\_complete的方式启动循环, 循环就会在完成原来注释的task之后退出, 动态添加的新的task就可能无法执行, 此时就可以使用run\_forever()的方式来启动循环, 使循环永不退出, 而在想要退出事件循环时, 就显式的调用loop.stop来结束循环的执行.

'''显示退出循环'''

import asyncio

import time

now = lambda: time.time()

# 1. 定义协程函数

async def task1():

print("task1 start")

await asyncio.sleep(5)

print("task1 done")

async def task2():

print("task2 start")

await asyncio.sleep(1)

print("task2 done")

def callback(future):

loop.stop()

start = now()

# 获取默认的事件循环对象

loop = asyncio.get\_event\_loop()

# 注册task

task1 = asyncio.Task(task1())

task2 = asyncio.Task(task2())

# 给task2添加了结束循环的callback函数, task2执行完毕后，就会立刻关闭循环

task2.add\_done\_callback(callback)

# 启动事件循环：

# 而loop.stop会在run\_forever的合适时候退出.

loop.run\_forever()

# 如果使用loop.stop()方法来结束run\_until\_complete的运行, 由于还有任务没完成，stop后，run\_until\_complete会抛出异常, 所以loop.stop()一般不用来结束run\_until\_complete启动的循环, 而只用来结束run\_forever启动的循环.

# loop.run\_until\_complete(task)

print("耗时:", now()-start)

运行程序, 因为task2只等待1s, 而task2运行完成后就会调用loop.stop()来结束循环, 所以整个程序的运行时间为1s多一点.

##### 例9：将event loop放在子线程中运行

例1到例8中, 因为只有一个线程, 所以event loop是在主线程中运行的. 在执行到loop.run\_until\_complete时, 主线程会阻塞, 直到所有的task执行结束才会结束阻塞继续执行下面的代码. 在执行到loop.run\_forever()时, 也会阻塞, 直到调用loop.stop()方法时才会结束主线程的阻塞, 继续执行下面的代码. 为了使主线程能够继续向下执行其它的操作, 就需要把event loop放在子线程中运行.

'''将event loop放在子线程中运行'''

import asyncio

import time

import threading

now = lambda :time.time()

# 定义一个协程函数

async def task\_func():

print("task\_func: ", threading.current\_thread().name)

print("task\_func start")

await asyncio.sleep(3)

print("task\_func done")

# 定义start\_loop普通函数, 接收一个loop循环, 使用run\_forever方式启动循环.

# start\_loop方法是在子线程中运行的.

def start\_loop(loop):

# 设置一个事件循环, 把loop绑定到当前的事件循环中. 因为是在子线程中执行的start\_loop()方法, 所以这里的loop会绑定到新的线程中.

asyncio.set\_event\_loop(loop)

loop.run\_forever()

# callback为协程函数task\_func的回调函数

def callback(t):

print("callback: ", threading.current\_thread().name)

time.sleep(t)

print("callback done")

start = now()

# 之前是以asyncio.get\_event\_loop()的方法来创建一个事件循环的. 此时以new\_event\_loop()方法来创建一个循环. loop是存在于当前线程中的唯一的一个事件循环对象, 如果使用get\_event\_loop()方法创建的loop, 就会绑定当前的线程, 也就是主线程. 如果想要创建出一个在其它线程中使用的loop, 就要使用new\_event\_loop()方法创建出一个全新的事件循环对象.

new\_loop = asyncio.new\_event\_loop()

# 在子线程中启动事件循环. 调用start\_loop方法, 把创建的new\_loop作为参数传递到start\_loop中, 则new\_loop就会随着start\_loop方法在子线程中启动并运行.

t = threading.Thread(target=start\_loop, args=(new\_loop,))

t.start()

# 向子线程中运行的事件循环中注册事件. 因为要向子线程中事件循环中注册事件, 所以此时就不能使用之前的三种方法来启动事件循环了. 这三种方法是向当前的线程也就是主线程中运行的事件循环中注册事件. 所以要使用asyncio.run\_coroutine\_threadsafe方法, 指定协程对象和要把协程对象注册到哪个循环中. 这样, task\_func()这个协程对象就会向new\_loop这个循环中运行.

# task = loop.create\_task(coroutine)

# task = asyncio.ensure\_future(coroutine)

# task = asyncio.Task(coroutine)

asyncio.run\_coroutine\_threadsafe(task\_func(), new\_loop)

# call\_soon，立刻安排回调函数并执行. 想要给某个task添加一个回调函数, 就要使用创建出来的task的add\_done\_callback方法来指定回调函数. 如 task.add\_done\_callback(callback). 而使用loop.call\_soon()时, 会立刻调用回调函数并执行.

new\_loop.call\_soon(callback, 1)

# 同call\_soon，但它是线程安全的, 当出现资源竞争等问题时, 会更安全一些.

new\_loop.call\_soon\_threadsafe(callback, 2)

# 主线程后面可以继续执行其他代码

# 但在最后也会挂起，等待子线程的结束

print("主线程：可以去做其他事情")

print("主线程：可以去做其他事情")

print("主线程：可以去做其他事情")

运行程序, 此时的task\_func和callback函数都是在子线程中执行的, 也就是在事件循环所在的那个子线程中运行的. 而主线程就可以空闲出来运行其它的代码了.

但是一般情况下更推荐在主线程中运行事件循环. 因为如果把一些方法或类用到子线程中时, 可以会导致代码的运行出现一些意想不到的问题.

#### asyncio爬虫应用

* aiohttp
* asyncio + requests/urllib

#### aiohttp

aiohttp是一个基于asyncio的Python实现的HTTP客户端/服务端库.

爬虫中则是使用aiohttp模块的客户端功能. 使用此模块来所发起的请求就是异步的并发请求了.

[aiohttp官方文档](https://aiohttp.readthedocs.io/en/stable/)

<https://aiohttp.readthedocs.io/en/stable/>

用法：

import aiohttp

import asyncio

async def fetch(session, url):

async with session.get(url) as response:

return await response.text()

async def main():

async with aiohttp.ClientSession() as session:

html = await fetch(session, 'http://www.baidu.com')

print(html)

loop = asyncio.get\_event\_loop()

loop.run\_until\_complete(main())

#### asyncio + requests/urllib

但要在aiohttp中结合requests, urllib模块来使用, 就要需要从头学习aiohttp模块, 手动实现代码, 学习成本会变高, 所以还是使用asyncio模块来实现requests/urllib的爬虫代码.

requests/urllib模块由于与asyncio模块兼容性差，因此很难将其它模块与asynco模块配合起来使用, 可以将task对象利用子线程/线程池来执行, 把得到的结果再返回到主线程/进程中使用协程进行处理.

import asyncio

import requests

async def main():

loop = asyncio.get\_event\_loop()

# 使用run\_in\_executor方法把爬虫代码放到一个线程池或进程池中运行.

future1 = loop.run\_in\_executor(None, requests.get, 'http://www.baidu.com')

future2 = loop.run\_in\_executor(None, requests.get, 'http://www.taobao.com')

# 因为main本身是一个协程函数, 它本身又await了两个future, 相当于嵌套了2个协程函数, 所以这里的future1和future2是同步的执行. 想要实现future1, future2的异步执行, 就要像例6中的那样, 使用asyncio.wait(tasks), asyncio.as\_completed(tasks), asyncio.gather(\*tasks)方法来发起多个异步的请求.

response1 = await future1

response2 = await future2

print(response1.text)

print(response2.text)

loop = asyncio.get\_event\_loop()

loop.run\_until\_complete(main())

### 4.2. gevent与爬虫

#### Gevent模块

gevent是一个基于协程的Python网络库，它使用 greenlet在libev或libuv事件循环之上提供高级同步API. 可以使用gevent来实现单线程中的异步并发.

Gevent原理：

* libev + libuv(Reactor模型)：Reactor模型是基于第3方的libev+libuv库来实现的. gevent是调用了这两者的接口来实现的事件循环. gevent内部的事件循环用于监听I/O事件的是否就绪/完成。
* greenlet：使用greenlet来实现任务的切换, greenlet的实现原理与yield类似, 但是是比yield更彻底的协程切换。通过栈的完全复制切换来实现不同协程之间的切换。

##### 例1：greenlet切换协程与yield切换协程

# 协程1和协程2属于同级并列关系，不是嵌套

def greenlet\_test():

# 单独使用的greenlet, 所以导入的是greenlet

from greenlet import greenlet

# 创建了test1和test2这两个任务函数

def test1():

print("test1 start")

# 执行协程2：在协程1内部直接switch切换到协程2执行. 在协程2中又会切换到协程1中执行, 切换回来时, 会从上次切换到协程2的地方中继续执行代码.

corotine2.switch()

print("test1 done")

# 执行协程2：在协程1内部直接switch切换到协程2执行

corotine2.switch()

def test2():

print("test2 start")

# 执行协程1：在协程2内部直接switch切换到协程1执行

corotine1.switch()

print("test2 done")

# 使用greenlet创建出2个协程对象

corotine1 = greenlet(test1)

corotine2 = greenlet(test2)

# 使用corotine.switch来执行某个协程.

# 执行协程1. 在协程1中执行到特定的代码时手动切换到协程2, 执行协程2时再手动切换到协程1, 此时在协程1中会继续接着执行上次切换到功程2中时的代码.

# 如果按照类似于线程的方式来对协程进行划分, corotine1.switch()所在的协程就是 "主协程", 调用主协程的switch方法, 就会切换到协程1中执行. 而在协程1和协程2中也可以切换到另一个协程中, 也就是协程可以实现轮转切换.

corotine1.switch()

# python的协程的切换方式. 使用yield实现了2个最简单的协程.

def yield\_test():

def test1():

print("test1 start")

# 因为是使用的主协程调用的协程1, 所以执行到yield时, 会把控制权交给主协程.

yield

print("test1 done")

def test2():

print("test2 start")

yield

print("test2 done")

# 创建了两个协程对象, 实际上是生成器对象

corotine1 = test1()

corotine2 = test2()

# 执行协程1, 执行到yield时会把程序的控制权交出来, 继续向下执行代码.

# 在主协程中调用next()方法来执行协程1, 控制权由主协程交给协程1

next(corotine1)

# 执行协程2, 控制权由主协程交给协程2. 即使用python中的yield进行协程的切换时从哪个协程切换过来的, 只能切换到那个协程中. 而greenlet中就可以在不同的协程中自由进行切换. 所以greenlet以及基于greenlet的gevent中协程的切换是更彻底的切换

next(corotine2)

try:

# 执行协程1

next(corotine1)

# 如果生成器中没有元素时, 调用next方法会抛出一个StopIteration的异常.

except StopIteration:

pass

try:

# 执行协程2

next(corotine2)

except StopIteration:

pass

if \_\_name\_\_ == '\_\_main\_\_':

greenlet\_test()

print("\*\*\*\*\*\*\*\*")

yield\_test()

'''

test1 start

test2 start

test1 done

test2 done

\*\*\*\*\*\*\*\*

test1 start

test2 start

test1 done

test2 done

'''

##### 例2：greenlet与gevent

gevent中greenlet工作的原理.

import gevent

import time

now = lambda: time.time()

def test1():

print("test1 start")

gevent.sleep(1) # 模拟IO事件，执行到此处时，gevent会自主交出当前协程的执行权. 一旦把test1和test2中的gevent.sleep()都注释掉, 代码就会变成同步的了, 会先执行完test1中的所有代码, 再去执行test2中的所有代码

print("test1 done")

def test2():

print("test2 start")

gevent.sleep(1) # 模拟IO事件，执行到此处时，gevent会自主交出当前协程的执行权

print("test2 done")

if \_\_name\_\_ == '\_\_main\_\_':

start = now()

# 创建gevent的greenlet协程对象, 不同与之前的原生的greenlet对象.

gevent\_greenlet1 = gevent.spawn(test1)

gevent\_greenlet2 = gevent.spawn(test2)

# 使用gevent.joinall来注册所有的gevent\_greenlet协程对象.

gevent.joinall([

gevent\_greenlet1,

gevent\_greenlet2,

])

print(now()-start)

例2分析：

理论上，greenlet之间的调度需要直接调用switch实现，但gevent中的greenlet的switch方法会由gevent主动调用.

原因：gevent会自主侦测(asyncio中需要主动使用await)，一旦发现I/O，便立刻调用greenle的switch方法 gevent的joinall方法就相当于将gevent的greenlet对象进行注册，并全部进行自主监听和调度.

gevent.sleep就是一个类似I/O的信号，可被gevent侦测到，因此可借助其，在gevent中实现非I/O任务的协程切换

##### 例3：猴子补丁

猴子补丁的作用：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，这样gevent才能让gevent能自主检测到他们，从而帮助gevent实现程序的异步运行

import gevent

import urllib.request

# 打补丁

import gevent.monkey

# patch\_all() 会替换掉python标准库中所有IO模块. 如select, socket等.

gevent.monkey.patch\_all()

def test1():

print("test1 start")

# 网络I/O

# 打补丁前，无法被gevent侦测到

# 打补丁后，可以被gevent侦测到

urllib.request.urlopen("http://www.baidu.com")

print("test1 done")

def test2():

print("test2 start")

urllib.request.urlopen("http://www.baidu.com")

print("test2 done")

if \_\_name\_\_ == '\_\_main\_\_':

g1 = gevent.spawn(test1)

g2 = gevent.spawn(test2)

gevent.joinall([

g1,

g2,

])

打补丁前为什么没有自动进行IO的切换呢, gevent是自动对IO请求进行侦测和判断的, 但是无法自动识别第三方模块中的IO请求, 想要使用gevent来实现IO请求的自动切换, 就需要打上补丁. 把python标准库中的socket模块替换为gevent中实现的异步的socket模块, 现在urllib中使用的socket就是gevent中的socket模块了, 就能够实现网络IO请求的自动判断及协程的自动切换.

'''运行结果：打补丁前，是同步的

test1 start

test1 done

test2 start

test2 done

'''

'''运行结果：打补丁后，是异步的

test1 start

test2 start

test2 done

test1 done

'''

##### 例4：gevent协程池

from gevent.pool import Pool

import urllib.request

# 打补丁

import gevent.monkey

gevent.monkey.patch\_all()

def spider(url):

# 打印当前协程的名称. gevent.getcurrent().name类似于threading.current\_thread().name

print("Coroutine(%s): spider[%s] start"%(gevent.getcurrent().name,url))

urllib.request.urlopen(url, timeout=2)

print("Coroutine(%s): spider[%s] done"%(gevent.getcurrent().name,url))

if \_\_name\_\_ == '\_\_main\_\_':

urls = [

"https://www.baidu.com",

"https://www.google.com", # 因为设置了urllib.request.urlopen中的timeout为2s, 超时2s就会抛出异常, 默认情况下google.com会抛出异常.

"http://www.jd.com",

"http://www.taobao.com",

"https://www.tencent.com"

]

# 创建基于gevent的协程池. 这里的协程池与python标准库中的进程池/线程池的使用方法和效果几乎是完全相同的.

pool = Pool(3)

for url in urls:

pool.apply\_async(spider, args=(url,))

pool.join()

'''运行结果：

Coroutine(Greenlet-0): spider[https://www.baidu.com] start

Coroutine(Greenlet-1): spider[https://www.google.com] start

Coroutine(Greenlet-2): spider[http://www.jd.com] start

Coroutine(Greenlet-2): spider[http://www.jd.com] done

Coroutine(Greenlet-2): spider[http://www.taobao.com] start

Coroutine(Greenlet-0): spider[https://www.baidu.com] done

Coroutine(Greenlet-0): spider[https://www.tencent.com] start

Coroutine(Greenlet-0): spider[https://www.tencent.com] done

Coroutine(Greenlet-2): spider[http://www.taobao.com] done

......后面是异常信息(未贴出)

'''

**注意：** 由于google.com一般情况是无法访问的，同时由于设置了timeout，所以会导致请求google.com的那个greenlet，会出现异常而退出。

如果执行google.com请求的协程是1号协程, 会抛出异常, 1号协程就不会正常向下执行了, 这个协程会永久性的挂掉, 无法再次使用. 但发生异常的协程不会影响其它协程的正常执行, 其它协程还会正常执行代码, 直到其它所有的协程都执行完成后才最终抛出异常. 所以在使用gevent中的协程时, 最好也像使用python中的进程池/线程池时一样, 使用try…except…来捕获异常并对异常进行处理,

这些特征都非常类似多线程或多进程，可以看出gevent的程序设计风格，非常符合利用多线程、多进程的设计风格；其实你完全可以把greenlet就当做线程来看待，同时gevent的Pool的API和进程池、线程池的几乎一样，使用起来非常方便。

但要注意的是，gevent是协程，并且只能调度网络I/O，而且为了和其他网络模块进行合作，如reqeusts、urllib、python-redis等，一般都必须打猴子补丁，这也有可能会给gevent带来一些兼容性的问题。

### 4.3. tornado与爬虫

注意：本章节所讲tornado版本为当前最新的5.0+版本，因为新版中有比较多特性，因此大家学习前先，注意自己安装的版本

#### tornado

tornado定位其实本是一个web框架，并且拥有一个基于I/O多路复用与python的协程实现的异步功能，其实其本质上也可以看做是Reactor模式设计的事件驱动异步模型。

tornado自身实现了一个事件循环, 与asyncio的事件循环库的代码不太一样, 但原理和api都是类似的，当前5.0+版本的tornado的异步代码同asyncio的代码非常相似，而且你完全可以把tornado的异步代码就当做是asyncio那样的方式去使用和开发。[tornado的github地址](https://github.com/tornadoweb/tornado)

[**tornado.ioloop**](http://www.tornadoweb.org/en/stable/ioloop.html#module-tornado.ioloop): tornado内部实现的事件循环库，内部多数API同asyncio事件循环的功能一致。http://www.tornadoweb.org/en/stable/ioloop.html#module-tornado.ioloop

同时tornado中实现协程函数已经支持直接使用async/await来定义，此前是使用torando.gen.coroutine装饰器与yield来实现，类似asyncio.coroutine装饰器与yield。

如果你倾向与使用asyncio来进行异步编程，那么现在新版本的tornado将是非常好的一个帮手，它里面封装了很多很强大的模块可直接使用。其中之一，就是接下来要讲到的tornado.httpclient模块。

#### tornado.httpclient模块

<http://www.tornadoweb.org/en/stable/http.html>

[**tornado.httpclient**](http://www.tornadoweb.org/en/stable/http.html)是tornado中实现完成的一个具备**异步的高性能HTTP客户端库**，因此你完全可以使用它来代替其他的如urllib、requests、aiohttp等模块来实现爬虫，甚至在笔者看来，这是一个更好的方案。原因: 1. 自身实现了协程, api比aiohttp模块好用好理解, 使用起来更方便. 2. tornado.httpclient中封装了HTTPRequest和HTTPResponse两个对象, 功能也比较完善, 可以直接使用其来实现爬虫的代码. 3. tornado的http功能丰富, 封装完善, 如tornado.httputil模块, 可以用来处理http请求头和url. 4. 性能高. 实现高并发的网络IO处理.

tornado.httpclient底层有两种实现方式：

* [tornado.simple\_httpclient.SimpleAsyncHTTPClient](https://github.com/tornadoweb/tornado/blob/master/tornado/simple_httpclient.py)：默认方式。tornado基于内部的TCPClient实现的简易HTTP异步客户端。功能相对简单.
* [tornado.curl\_httpclient.CurlAsyncHTTPClient](https://github.com/tornadoweb/tornado/blob/master/tornado/curl_httpclient.py)：依赖libcurl和pycurl，通过python利用pycurl调用libcurl接口实现的HTTP异步客户端。相对于SimpleAsyncHTTPClient多了不少功能，且性能更强，推荐使用。调用操作系统中的curl命令的接口实现的http异步客户端. 由于curl是基于c语言编写的, 所以处理速度更快. 功能更加完善, 如使用代理, 推荐使用.

使用方式:

AsyncHTTPClient.configure("tornado.curl\_httpclient.CurlAsyncHTTPClient")

用法：

例1：同步版tornado的HTTP客户端

from tornado import httpclient

http\_client = httpclient.HTTPClient()

try:

response = http\_client.fetch("http://www.baidu.com/")

print(response.body)

except httpclient.HTTPError as e:

# HTTPError is raised for non-200 responses; the response

# can be found in e.response.

print("Error: " + str(e))

except Exception as e:

# Other errors are possible, such as IOError.

print("Error: " + str(e))

http\_client.close()

例2：异步版tornado的HTTP客户端

from tornado import httpclient, ioloop

# 定义一个协程函数

async def spider():

http\_client = httpclient.AsyncHTTPClient()

try:

# fetch返回的是一个future对象

response = await http\_client.fetch("http://www.baidu.com")

except Exception as e:

print("Error: %s" % e)

else:

print(response.effective\_url)

if \_\_name\_\_ == '\_\_main\_\_':

# 获取当前线程的事件循环，类似asyncio.get\_event\_loop

io\_loop = ioloop.IOLoop.current()

# 运行spider协程，直到运行完毕后就退出循环，类似asyncio\_loop.run\_until\_complete

# run\_sync方法中只要传入协程函数, 就会自动调用http\_client.fetch方法来获取数据. 如果有多个爬虫的请求, 就可以使用await asyncio.wait(tasks), asyncio.as\_completed(tasks), await asyncio.gather(\*tasks) 方法来实现多个异步请求的发送. 比asyncio中实现的方法更加简洁明了.

io\_loop.run\_sync(spider)

#### tornado.httpclient.HTTPRequest与tornado.httpclient.HTTPResponse

httpclient的fetch方法即可以接收一个url地址，也可以接收一个[HTTPReques](http://www.tornadoweb.org/en/stable/httpclient.html#tornado.httpclient.HTTPRequest)对象，返回的结果是一个[HTTPResponse](http://www.tornadoweb.org/en/stable/httpclient.html#tornado.httpclient.HTTPResponse)对象

如果使用tornado.httpclient客户端发请求，这两个对象的使用必须重点掌握。

### 4.4. twisted与爬虫

#### twisted模块

##### What is Twisted?

Twisted is an event-driven networking engine written in python and licensed under the open source licence.

安装：

pip install twisted

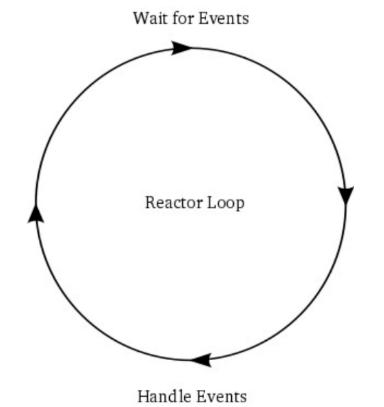
pip install twisted[tls] # 安装支持HTTPS的twisted

twisted其实是最名副其实的基于Reactor模式而设计的事件驱动异步网络库。可以实现单线程的异步并发.

twisted可以说是更贴近传输层一个异步网络库，但是它的代码风格比较特殊，上手较难。

##### twisted核心功能：reactor

twisted的核心是reactor，其实也就是twisted中的io loop事件循环, 或是asyncio中的event loop。



整个事件循环就是在单线程中运行的循环, 在循环的过程中, 一端等待着事件的触发, 当事件触发之后, 另一端就会对事件进行捕获和处理, 一个事件完成后就再次进行循环, 等待着事件的触发, 如果捕获到了事件, 就再次进行处理, 再次等待事件触发, 如此循环下去.

twisted也是利用reactor模式和python的协程实现。reactor的event loop可以看做是协程的调度者（包括前面的其他模块）。但不同的是，twisted的异步代码中，看不到协程的影子，就是定义的一个普通函数, 因为其中协程的定义和调度由twisted负责完成。

##### 例1：reactor启动与退出

import time

from twisted.internet import reactor

now = lambda: time.time()

# 定义一个普通的函数, 相当于task函数.

def task(x):

print('The Number Is :', x)

time.sleep(0.1)

if x == 9:

# 退出事件循环

reactor.stop()

start = now()

# 添加10个task任务.

# 但是由于task中没有涉及到网络问题, 也没有按照twisted的异步任务规则来定义task任务函数, 所以这里的10个任务会以同步的方式运行. 相当于逐一把10个任务添加到事件循环中, 处理完一个, 再添加并处理下一个. twisted不支持类似于asyncio中定义协程函数的方法, 想要在twisted中实现异步, 所实现的任务必须是一个网络任务.

for i in range(10):

# 当事件循环被启动时, 会自动调用task任务函数, 并且把i作为参数传递进去.

reactor.callWhenRunning(task, i)

# 启动事件循环, 调用task任务函数.

reactor.run()

print("耗时:", now()-start)

'''运行结果：同步结果

The Number Is : 0

The Number Is : 1

The Number Is : 2

The Number Is : 3

The Number Is : 4

The Number Is : 5

The Number Is : 6

The Number Is : 7

The Number Is : 8

The Number Is : 9

耗时: 1.0598435401916504

'''

##### twisted抽象层之Transports对象

twisted是一个网络库, 主要实现的功能是对网络协议进行异步操作. transports对象代表的是每一种网络协议底层的传输层.

一个Twisted的Transport对象具体代表一个可以收发字节的单条连接。其实也就是代表一个TCP或UDP的套接字连接的抽象，每一次数据的收发就是一次具体的I/O操作。因此twisted其实就是通过利用I/O多路复用控制每一个套接字的每次数据收发操作，实现的异步I/O，这也是前面几个模块(asyncio、tornado等)的底层原理，只不过twisted将其开放出来，可供我们直接控制。

但是一般情况下，我们不会自己实现一个Transport，而是直接使用twisted已经帮我们实现的类。

##### twisted抽象层之Protocols对象

Protocols对象代表的就是一个Transports对象的连接所对应的协议的抽象，比如HTTP、FTP或自定义协议等协议，为一个具体的连接提供协议解析。

程序每建立一条连接，都需要一个协议实例。也就是说，每一个Transports对象都会有一个Protocols对象。

Twisted内部已经内置了许多Protocols协议对象，如后面将用到封装了HTTP协议的Protocols对象。

##### twisted抽象层之Protocol Factories

由于我们可以自己定义Protocol类，并且要将其交给twisted来控制使用，因此twisted需要一种方式来为一个新的连接创建一个合适的协议。这就是Protocol Factories的作用。在protocol factories中定义一些属性或方法, 告诉twisted如何来使用protocols对象.

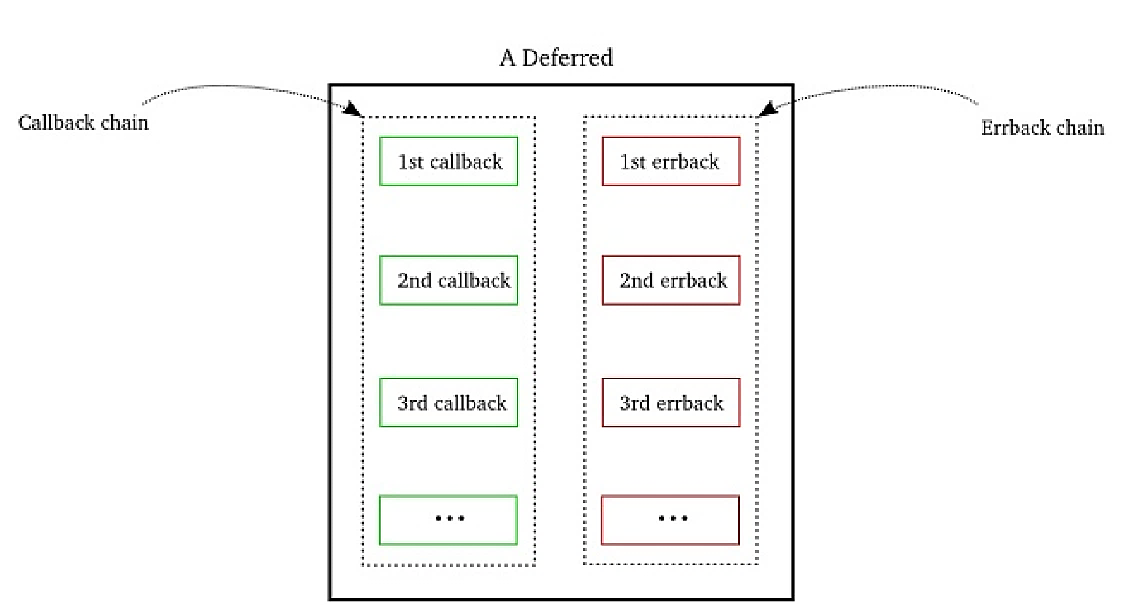
总的来说, transports代表的是tcp或udp的连接, 而protocols代表了连接对应的应用层的协议, 负责把连接传递的数据按照协议转换为对应的文本信息, protocol factories就是告诉twisted如何来构建出protocol对象并使用的.

以http的异步来进行说明, twisted, asyncio, tornado等异步库, 实现异步时都不是以http的一个连接作为单位的, 而是以tcp或udp每一次的收数据或发数据的过程为单位来控制异步IO, 从而实现异步功能的. 以下载一个几百K的页面数据的过程为例进行说明, 在传输层真正发送数据的时候, tcp是把数据分成很多段, 每次发送一段数据, 每一次发送数据都是一个IO操作, twisted通过以上三个对象对外开放了对每一次IO操作控制的接口, 程序员可以通过构建以上三个对象来实现对每一次IO操作的控制. 而asyncio, tornado等库则没有开放类似操作的接口, 只是在整个http的角度来看时实现的是异步的操作.

##### twisted中的回调：Deferred

由于回调在异步程序中大量被使用，并且正确的使用这一机制需要一些技巧。因此，Twisted开发者设计了一种抽象机制Deferred-让程序员在使用回调时更加简便。

在reactor模式中, 在注册一个事件时, 需要有一个事件, 同时有这个事件对象的事件处理器, 事件处理器通常会包括两种类型, 即事件成功时和失败时的处理器, 而twisted就是基于类似的机制来实现回调的. 当事件成功时, 会执行成功的回调函数, 当事件执行失败时, 会执行失败的回调函数, 并且这里还可以添加链式的回调函数, 也就是说可以设置多个回调函数, 依次的进行调用. Defferred就是twisted中用来控制程序异步回调的功能. 同时, 错误的回调函数不只是在事件执行失败时可以进行调用, 在调用事件成功时成功的回调函数时, 如果其中出现了异常或错误, 也可以立刻调用错误的回调函数.



下面我们就结合twisted.web.client模块熟悉这些概念

#### twisted.web.client模块

twisted.web.client.Agent是twisted中实现的具备异步功能的HTTP客户端类，可用来发起客户端请求。

##### 例2：Agent与Defered对象使用

from twisted.internet import reactor # 事件循环

from twisted.web.client import Agent # 相当于客户端

from twisted.web.http\_headers import Headers # 请求头, 响应头的设置

# 创建agent对象, 使用Agent传入reactor这个循环对象

agent = Agent(reactor)

# agent.request返回一个defered对象，用于设置回调函数.

# defered类似于asyncio中的task任务对象. 创建了task对象后, 可以使用task.add\_done\_callback()来添加回调函数.

defered = agent.request(

b'GET', # 请求方法, 必须是bytes类型的

b'http://www.baidu.com/s?wd=python', # 请求url, 必须是bytes类型的

Headers({'User-Agent': ['Twisted Web Client Example']}), # 请求头

None) # 请求体

def successCallback(response):

print('Response received：', response)

# 设置成功回调：如果任务执行成功，那么将会调用该回调函数

defered.addCallback(successCallback)

def errorCallback(error):

print("errback: ", str(error))

# 设置失败回调：如果任务执行中出现异常或成功回调函数出现异常，那么将会调用失败回调

defered.addErrback(errorCallback)

def callbackShutdown(ignored):

# 类似于asyncio模块中的run\_forever, 必须要手动显示调用reactor.stop(), 否则循环会一直执行.

reactor.stop()

# 设置最终回调：无论成功和失败，都必然会调用的回调函数.

defered.addBoth(callbackShutdown)

# addCallback/addErrback/addBoth的逻辑类似于try/except/finally

# 启动事件循环, 此时才会真正的把agent.request中构建的请求发送出去, 获取响应.

reactor.run()

##### 例3：Agent发起post请求

from twisted.internet import reactor

from twisted.web.client import Agent

from twisted.web.http\_headers import Headers

'''创建一个数据生产者，实现POST请求'''

# 构造请求体的方法. 固定写法.

from zope.interface import implementer

from twisted.internet.defer import succeed

from twisted.web.iweb import IBodyProducer

@implementer(IBodyProducer)

class BytesProducer(object):

# 实例化时需要传入body,

def \_\_init\_\_(self, body):

self.body = body

self.length = len(body)

def startProducing(self, consumer):

consumer.write(self.body)

return succeed(None)

def pauseProducing(self):

pass

def stopProducing(self):

pass

# 创建agent对象

agent = Agent(reactor)

# agent.request返回一个defered对象，用于设置回调函数

defered = agent.request(

b'POST', # 请求方法

b'http://www.baidu.com/s?wd=python', # 该接口此处无效，它只处理get请求

Headers({'User-Agent': ['Twisted Web Client Example']}),

BytesProducer('请求体'.encode()) # 请求体, 这里的请求体不能是普通的字符串或者bytes类型的字符串, 必须利用BytesProducer转换数据, 把请求体的内容作为参数传递到BytesProducer中进行实例化, 生成一个对象, 作为请求体的内容.

)

def successCallback(response):

print('Response received：', response)

# 设置成功回调：如果任务执行成功，那么将会调用该回调函数

defered.addCallback(successCallback)

def errorCallback(error):

print("errback: ", str(error))

# 设置失败回调：如果任务执行中出现异常或成功回调函数出现异常，那么将会调用失败回调

defered.addErrback(errorCallback)

def callbackShutdown(ignored):

reactor.stop()

# 设置最终回调：无论成功和失败，都必然会调用的回调函数

defered.addBoth(callbackShutdown)

# addCallback/addErrback/addBoth的逻辑类似于try/except/finally

# 启动事件循环

reactor.run()

##### 例4：使用自定义协议对象异步地接收响应体

例3中的代码只是实现了twisted发送请求并获取响应的功能, 还需要对响应的内容进行解析. 在例2中, 打印出的response对象是

<twisted.web.\_newclient.Response object >

查看twisted api, <https://twistedmatrix.com/documents/current/api/moduleIndex.html>

搜索twisted.web.\_newclient, 查看twisted.web.\_newclient中的Response,

<https://twistedmatrix.com/documents/current/api/twisted.web.client.Response.html>

Response对象的结构如下:

def \_\_init\_\_(self, version, code, phrase, headers, \_transport): [(source)](https://github.com/twisted/twisted/tree/twisted-18.9.0/src/twisted/web/_newclient.py#L1038)

|  |  |  |
| --- | --- | --- |
| Parameters | version | HTTP version components protocol, major, minor. E.g. (b'HTTP', 1, 1) to mean b'HTTP/1.1'. HTTP版本号 |
| code | HTTP status code. (type: [int](http://docs.python.org/library/stdtypes.html#int)) HTTP的状态码的数值 |
| phrase | HTTP reason phrase, intended to give a short description of the HTTP status code. HTTP状态码的文字描述 |
| headers | HTTP response headers. (type: [twisted.web.http\_headers.Headers](https://twistedmatrix.com/documents/current/api/twisted.web.http_headers.Headers.html)) 响应头 |
| \_transport | The transport which is delivering this response. |

def setPreviousResponse(self, previousResponse): [(source)](https://github.com/twisted/twisted/tree/twisted-18.9.0/src/twisted/web/_newclient.py#L1084)

*from*[*twisted.web.iweb.IResponse*](https://twistedmatrix.com/documents/current/api/twisted.web.iweb.IResponse.html#setPreviousResponse)

Set the reference to the previous [IResponse](https://twistedmatrix.com/documents/current/api/twisted.web.iweb.IResponse.html).

The value of the previous response can be read via [IResponse.previousResponse](https://twistedmatrix.com/documents/current/api/twisted.web.iweb.IResponse.html#previousResponse).

def deliverBody(self, protocol): [(source)](https://github.com/twisted/twisted/tree/twisted-18.9.0/src/twisted/web/_newclient.py#L1088)

*from*[*twisted.web.iweb.IResponse*](https://twistedmatrix.com/documents/current/api/twisted.web.iweb.IResponse.html#deliverBody)

Dispatch the given [IProtocol](https://twistedmatrix.com/documents/current/api/twisted.internet.interfaces.IProtocol.html) depending of the current state of the response.

所以response对象中并没有类似于requests模块中的content或text的方法, 所以这里的response并不是requests模块中类似的包含所有响应数据的响应. 使用twisted模块发送请求之后, 获取的response响应中只包含了协议和响应头, 并没有真正的开始接收响应体的数据. 使用deliverBody方法, 传入一个twisted协议对象protocol, 才能真正获取响应体的数据, protocal的作用是把字节数据转换为文本数据.

from twisted.internet import reactor

from twisted.internet.defer import Deferred, DeferredList

from twisted.internet.protocol import Protocol

from twisted.web.client import Agent

from twisted.web.http\_headers import Headers

# 为了使用deliverBody的方法来获取响应体的数据, 需要传入protocol协议对象, 可以创建一个协议Protocol类，用来收集数据 (接收响应体的数据).

# 在类中定义dataReceived和connectionLost方法, 这两个方法是protocol的接口方法, 会重写父类Protocol中的同名方法

class BodyCollector(Protocol):

def \_\_init\_\_(self, finished, uri):

self.finished = finished # 实例化时需要传入finished对象, finished对象是外部定义的defered对象，用于数据接收完毕后，触发相应回调函数

self.uri = uri

# 因为数据是分段进行接收的, 想要获取完整的数据, 就需要对分段的数据进行拼接, 定义一个bytes类型的body, 用来拼接并保存完整的数据.

self.body = bytes()

def dataReceived(self, data):

'''套接字每接收到一次数据，该方法就会被调用一次, 如果数据分为很多段进行接收, 此方法就会被调用很多次'''

print("data\_uri: ", self.uri, data)

self.body += data # 收集所有的的响应数据

def connectionLost(self, reason):

'''建立起socket连接时, 会有一个transport对象, 当socket连接关闭后，该方法会被调用'''

print('已完成数据的收集:', reason.getErrorMessage())

# 触发对应的回调函数. finished对象是用来设置回调函数的, 当response响应体的数据接收完成, 或者出现异常, 连接被断开时, connectionLost方法就会被调用, 也就会调用finished对象中绑定的callback回调函数.

# 连接断开时, 会执行finished对象中定义的回调函数, 并把接收到的完整的响应体的内容body传递到回调函数中进行进一步的处理.

self.finished.callback(self.body)

agent = Agent(reactor)

def successCallback(response):

'''此时还没有开始响应体，但已经收到响应头的信息，如下：'''

# print('Response version:', response.version)

# print('Response code:', response.code)

# print('Response phrase:', response.phrase)

# print('Response headers:')

# print(pformat(list(response.headers.getAllRawHeaders())))

# finished是实例化的Deferred()对象, 用来设置回调函数.

finished = Deferred()

# deliverBody方法接收一个Protocol对象，用来异步地接收数据

response.deliverBody(BodyCollector(finished, response.request.absoluteURI))

def handleBody(body):

print("twisted: ", len(body))

# 给finished对象添加回调函数.

finished.addCallback(handleBody) # 当完成后，回调会在Protocol类中调用

def errorCallback(error):

print(response.request.absoluteURI, error)

# 给finished对象添加错误时的回调函数

finished.addErrback(errorCallback)

return finished

# 创建一个单独的get方法的diferred对象.

d = agent.request(

b'GET',

b"http://www.baidu.com",

Headers({'User-Agent': ['Twisted Web Client Example']}),

None)

# 给请求对象添加成功时的回调函数

d.addCallback(successCallback)

def callbackShutdown(ignored):

reactor.stop()

# 给请求对象添加无论成功还是失败都要执行的回调函数, 并在此回调函数中执行reactor.stop()方法来显示的关闭循环.

d.addBoth(callbackShutdown)

if \_\_name\_\_ == '\_\_main\_\_':

# 开始循环的执行.

reactor.run()

# 用urllib比较下，最终的数据长度

import urllib.request

print("urllib:", len(urllib.request.urlopen("http://www.baidu.com").read()))

执行程序, 可以看到数据是分段进行接收的. 想要获取到完整的数据, 就要对获取的数据进行拼接. 定义一个body属性来对接收到的分段的数据进行拼接. 通过与urllib.request方法获取的响应体的长度进行对比, 判断twisted方法获取的响应体是否完整.

##### 例5：使用DeferedList管理多个Defere对象，完成并发控制

'''

from twisted.internet import reactor

from twisted.internet.defer import Deferred, DeferredList

from twisted.internet.protocol import Protocol

from twisted.web.client import Agent

from twisted.web.http\_headers import Headers

# 创建一个协议Protocol类，用来收集数据

class BodyCollector(Protocol):

def \_\_init\_\_(self, finished, uri):

self.finished = finished # 外部定义的defered对象，用于数据接收完毕后，触发相应回调函数

self.uri = uri

self.body = bytes()

def dataReceived(self, data):

# 每接收到一次数据，该方法就会被调用一次

print("data\_uri: ", self.uri, data)

self.body += data # 收集所有的的响应数据

def connectionLost(self, reason):

# 当socket连接关闭后，该方法会被调用

print('已完成数据的收集:', reason.getErrorMessage())

# 触发对应的回调函数

self.finished.callback(self.body)

agent = Agent(reactor)

def successCallback(response):

# 此时还没有开始响应体，但已经收到响应头的信息，如下：

# print('Response version:', response.version)

# print('Response code:', response.code)

# print('Response phrase:', response.phrase)

# print('Response headers:')

# print(pformat(list(response.headers.getAllRawHeaders())))

finished = Deferred()

# deliverBody方法接收一个Protocol对象，用来异步地接收数据

response.deliverBody(BodyCollector(finished, response.request.absoluteURI))

def handleBody(body):

print("twisted: ", len(body))

finished.addCallback(handleBody) # 当完成后，回调会在Protocol类中调用

def errorCallback(error):

print(response.request.absoluteURI, error)

finished.addErrback(errorCallback)

return finished

d = agent.request(

b'GET',

b"http://www.baidu.com",

Headers({'User-Agent': ['Twisted Web Client Example']}),

None)

# 这里如果stop退出循环，可能导致d2的请求无法完成

# d.addCallback(successCallback)

#

# def callbackShutdown(ignored):

# reactor.stop()

# d.addBoth(callbackShutdown)

# 添加第二个请求

d2 = agent.request(

b'GET',

b"https://www.taobao.com",

Headers({'User-Agent': ['Twisted Web Client Example']}),

None)

d2.addCallback(successCallback)

if \_\_name\_\_ == '\_\_main\_\_':

reactor.run()

'''

from twisted.internet import reactor

from twisted.internet.defer import Deferred, DeferredList

from twisted.internet.protocol import Protocol

from twisted.web.client import Agent

from twisted.web.http\_headers import Headers

# 创建一个协议Protocol类，用来收集数据

class BodyCollector(Protocol):

def \_\_init\_\_(self, finished, uri):

self.finished = finished # 外部定义的defered对象，用于数据接收完毕后，触发相应回调函数

self.uri = uri

self.body = bytes()

def dataReceived(self, data):

'''每接收到一次数据，该方法就会被调用一次'''

print("data\_uri: ", self.uri, data)

self.body += data # 收集所有的的响应数据

def connectionLost(self, reason):

'''当socket连接关闭后，该方法会被调用'''

print('已完成数据的收集:', reason.getErrorMessage())

# 触发对应的回调函数

self.finished.callback(self.body)

agent = Agent(reactor)

def successCallback(response):

'''此时还没有开始响应体，但已经收到响应头的信息，如下：'''

# print('Response version:', response.version)

# print('Response code:', response.code)

# print('Response phrase:', response.phrase)

# print('Response headers:')

# print(pformat(list(response.headers.getAllRawHeaders())))

finished = Deferred()

# deliverBody方法接收一个Protocol对象，用来异步地接收数据

response.deliverBody(BodyCollector(finished, response.request.absoluteURI))

def handleBody(body):

print("twisted: ", len(body))

finished.addCallback(handleBody) # 当完成后，回调会在Protocol类中调用

def errorCallback(error):

print(response.request.absoluteURI, error)

finished.addErrback(errorCallback)

return finished

# 创建多个请求, 测试twisted的异步功能. 使用方法类似于asyncio模块中的方法.

\_ds= []

for url in [

b"http://www.baidu.com",

b"https://www.taobao.com",

b"https://www.jd.com",

b"https://www.tmall.com",

b"https://www.tencent.com",

b"https://www.douban.com",]:

# 创建出多个diferred请求对象

d = agent.request(

b'GET',

url,

Headers({'User-Agent': ['Twisted Web Client Example']}),

None)

d.addCallback(successCallback)

\_ds.append(d)

# 由于需要给多个diferred请求对象添加addBoth方法, 当全部回调函数执行完成后显示的调用reactor.stop()方法来退出循环, 所以这里定义一个DeferredList()对象, 对多个defered对象进行统一的管理.

dl = DeferredList(\_ds)

def callbackShutdown(ignored):

reactor.stop()

# 当dl中所有的diferred对象都处理完成后, 才会去调用callbackShutdown方法来关闭整个循环的运行.

dl.addBoth(callbackShutdown)

if \_\_name\_\_ == '\_\_main\_\_':

reactor.run()

执行程序, 可能会先接收几段某个请求的响应体的数据, 再去接收几段另一个请求的响应体的数据, 各个请求体的数据是以异步的方式进行接收的. 使用twisted, 就可以在tcp连接的水平上对每次接收到的数据进行处理.

##### 例6：HTTP连接池（长连接）

from twisted.internet import reactor

from twisted.internet.defer import Deferred, DeferredList

from twisted.internet.protocol import Protocol

from twisted.web.client import Agent, HTTPConnectionPool

from twisted.web.http\_headers import Headers

# 创建一个协议Protocol类，用来收集数据

class BodyCollector(Protocol):

def \_\_init\_\_(self, finished, uri):

self.finished = finished # 外部定义的defered对象，用于数据接收完毕后，触发相应回调函数

self.uri = uri

self.body = bytes()

def dataReceived(self, data):

'''每接收到一次数据，该方法就会被调用一次'''

# print("data\_uri: ", self.uri, data)

self.body += data # 收集所有的的响应数据

def connectionLost(self, reason):

'''当socket连接关闭后，该方法会被调用'''

print('已完成数据的收集:', reason.getErrorMessage())

# 触发对应的回调函数

print(self.transport)

self.finished.callback(self.body)

# HTTP连接池，实现HTTP长连接

connect\_pool = HTTPConnectionPool(reactor, persistent=True)

# 创建的agent中会自动的维护一定的连接数的连接池, 默认情况下是2个连接数的连接池

agent = Agent(reactor, pool=connect\_pool)

def successCallback(response):

'''此时还没有开始响应体，但已经收到响应头的信息，如下：'''

# print('Response version:', response.version)

# print('Response code:', response.code)

# print('Response phrase:', response.phrase)

# print('Response headers:')

# print(pformat(list(response.headers.getAllRawHeaders())))

finished = Deferred()

# deliverBody方法接收一个Protocol对象，用来异步地接收数据

response.deliverBody(BodyCollector(finished, response.request.absoluteURI))

def handleBody(body):

print("twisted: ", len(body))

finished.addCallback(handleBody) # 当完成后，回调会在Protocol类中调用

def errorCallback(error):

print(response.request.absoluteURI, error)

finished.addErrback(errorCallback)

return finished

# 连续发起了两个baidu的请求, 同一个域名解析出来的ip地址是相同的, 就可以使用连接池或长连接对tcp连接进行复用. 就可以基于连接池创建agent

\_ds = []

for url in [

b"http://www.baidu.com/s?wd=python",

b"http://www.baidu.com/s?wd=itcast"]:

d = agent.request(

b'GET',

url,

Headers({'User-Agent': ['Twisted Web Client Example']}),

None)

d.addCallback(successCallback)

\_ds.append(d)

dl = DeferredList(\_ds) # 统一管理多个defered对象

def callbackShutdown(ignored):

reactor.stop()

dl.addBoth(callbackShutdown)

if \_\_name\_\_ == '\_\_main\_\_':

reactor.run()

twisted连接池属性的修改

from twisted.web.client import HTTPConnectionPool

# 创建连接池, 如果想要修改最大的连接数并使修改生效, 需要传入persistent=True的参数

pool = HTTPConnectionPool(reactor, persistent=True)

# maxPersistentPerHost控制最大的连接缓存数，默认值是2，

pool.maxPersistentPerHost = 5

# cachedConnectionTimeout控制，长连接缓存时间，默认是240秒

pool.cachedConnectionTimeout = 600

# closeCachedConnections方法可以关闭所有缓存的连接

pool.closeCachedConnections()

#### twisted模块总结

twisted是一个功能非常强大，且重量级异步网络编程库，而且学习周期相比其他模块会更长。但twisted能做的事情远比你想象的要多，如果你已经有一定网络编程经验，并打算继续研究它，那么twisted是非常值得学习的。

[查看更多](https://twistedmatrix.com/documents/current/web/howto/client.html)关于twisted.web.client的介绍

https://twistedmatrix.com/documents/current/web/howto/client.html

[查看更多](https://twistedmatrix.com/documents/current/api/moduleIndex.html)twisted的API文档信息

https://twistedmatrix.com/documents/current/api/moduleIndex.html

### 4.5. celery与爬虫

前面几个模块实现的都是在单线程中对异步并发进行控制的模块. 而celery是建立在上面几个模块的基础上实现的一个分布式异步任务框架，拥有简单，灵活，可靠等特性。可以实现异步任务的收集和分发等功能.

celery可使用redis/rabbitmq作为它的任务队列，专注于实时处理，同时还支持任务调度。

理论上可以利用它来帮助我们实现任何需要异步的地方，比如爬虫中的请求并发

#### celery库

[github源码地址](https://github.com/celery/celery)

https://github.com/celery/celery

[官方文档地址](http://docs.celeryproject.org/en/latest/index.html)

http://docs.celeryproject.org/en/latest/index.html

**celery的实现原理：**利用redis/rabbitmq作为它任务队列，结合eventlet/gevent实现的异步，用于异步处理大批量消息。

celery是基于其它第3方模块实现异步功能的, 查看celery的github地址中的说明, 在Concurrency异步中, 可以基于eventlet或gevent来实现异步功能. eventlet是基于事件驱动实现的线程库. gevent是基于事件驱动实现的协程库

celery基础概念：

* task（任务）: task是celery负责调度和管理的对象，代表需要执行的一个函数或方法. 是worker执行的最小的工作单位.
* worker（工人）: worker是celery的工作进程，专门负责执行task。可以有多个worker.

celery中有一个基于redis/rabbitmq的任务队列, 创建的任务会首先进入到任务队列中, 然后再进行任务的分发. 因为是分布式的任务队列, 有很多个分布式的工作进程, 在celery中通常称为worker, 不同主机中的worker就可以从分布式的队列中获取task任务进行处理.

celery中的worker默认情况下是使用eventlet多线程来执行任务的, 也可以指定使用基于gevent的协程来执行task任务.

* broker: broker就是celery的任务队列，负责接收/缓存/分发task. 一个broker的任务队列中的任务可以分发给多个worker进行执行.

celery中支持的任务队列. 推荐使用RabbitMQ及Redis

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Status | Monitoring | Remote Control |
| RabbitMQ | Stable | Yes | Yes |
| Redis | Stable | Yes | Yes |
| Amazon SQS | Stable | No | No |
| Zookeeper | Experimental | No | No |

* result\_backend: 用于存储task运行后返回的结果. task一般会有一个返回值, 而celery是一个分布式的框架, 可以指定task的返回值保存的位置.
* [Task result backend settings](http://docs.celeryproject.org/en/latest/userguide/configuration.html#task-result-backend-settings)
* [Database backend settings](http://docs.celeryproject.org/en/latest/userguide/configuration.html#database-backend-settings) 存储到数据
* [RPC backend settings](http://docs.celeryproject.org/en/latest/userguide/configuration.html#rpc-backend-settings)
* [Cache backend settings](http://docs.celeryproject.org/en/latest/userguide/configuration.html#cache-backend-settings)
* [Redis backend settings](http://docs.celeryproject.org/en/latest/userguide/configuration.html#redis-backend-settings) 存储到redis数据库
* [Cassandra backend settings](http://docs.celeryproject.org/en/latest/userguide/configuration.html#cassandra-backend-settings)
* [Elasticsearch backend settings](http://docs.celeryproject.org/en/latest/userguide/configuration.html#elasticsearch-backend-settings) ES search数据库
* [Riak backend settings](http://docs.celeryproject.org/en/latest/userguide/configuration.html#riak-backend-settings)
* [AWS DynamoDB backend settings](http://docs.celeryproject.org/en/latest/userguide/configuration.html#aws-dynamodb-backend-settings)
* [IronCache backend settings](http://docs.celeryproject.org/en/latest/userguide/configuration.html#ironcache-backend-settings)
* [Couchbase backend settings](http://docs.celeryproject.org/en/latest/userguide/configuration.html#couchbase-backend-settings)
* [CouchDB backend settings](http://docs.celeryproject.org/en/latest/userguide/configuration.html#couchdb-backend-settings)
* [File-system backend settings](http://docs.celeryproject.org/en/latest/userguide/configuration.html#file-system-backend-settings) 存储到文件中
* [Consul K/V store backend settings](http://docs.celeryproject.org/en/latest/userguide/configuration.html#consul-k-v-store-backend-settings)

celery中最基础的两个概念：任务(task)、工人(worker)、broker

* 任务相当于一个交由celery进行调用的函数
* 工人是celery工作进程的，任务全部是由工人去完成。并且可以有启动多个worker。
* borker相当于任务的调度中心，任务是由broker

#### celery使用：

celeryconfig.py: 设置celery启动的基本配置，如任务队列、数据存储等配置项. 这里把队列和数据都保存到redis数据库中.

from \_\_future\_\_ import absolute\_import, unicode\_literals

import os

import sys

# 将当前路径，作为导包的第一路径

sys.path.insert(0, os.getcwd())

# ## Note: Start worker with -P gevent,

# do not use the worker\_pool option.

# 设置broker的信息. ip:port/db. 消息队列或任务队列的设置. 所以首先要运行redis容器. 这里的ip和port就基于redis容器的参数进行设置

broker\_url = 'redis://10.211.55.3:6379/0'

# 设置结果存储信息, 指定task任务的返回值保存的位置, 保存到redis的1号数据库中.

result\_backend = 'redis://10.211.55.3:6379/1/'

# 设置结果的过期时间. 如果经过这么长的时间还不进行处理的话, 就会自动进行删除.

result\_expires = 30 \* 60

# 设置含有task模块的模块名称. 因为下面是在task.py中定义并封装了task, 所以这里要写为模块的名称 "task".

imports = ('tasks',)

tasks.py: 注册task函数/方法

from \_\_future\_\_ import absolute\_import, print\_function, unicode\_literals

import requests

from celery import Celery

# 通过celery创建一个应用对象, 使用该对象来加载上面的配置文件.

app = Celery()

app.config\_from\_object('celeryconfig')

# 使用task装饰器，注册urlopen函数为task. 就可以把urlopen函数作为celery的一个task任务来使用了.

# ignore\_result如果为True，意味着celery不会把函数返回的结果保存到celeryconfig.py中定义的result\_backend中.

@app.task(ignore\_result=False)

def urlopen(url):

# urlopen函数中实现了发起网络请求的功能, 相当于爬虫的请求模块.

print('Opening: {0}'.format(url))

try:

response = requests.get(url)

except requests.exceptions.RequestException as exc:

print('Exception for {0}: {1!r}'.format(url, exc))

return "fail", url

print('Done with: {0}'.format(url))

# 返回的结果必须是能被json序列化的数据

# 如unicode类型字符串、数值、含有unicode类型字符串或数值的字典、列表等

return {"url": url}

spider.py：自定义的爬虫程序(使用注册的task函数)

在task.py模块中只是定义了urlopen的task, 想要使用, 就需要在spider中引入.

import time

import tasks

now = lambda: time.time()

def use\_celery():

'''通过delay方法传参调用，才会将函数交由celery进行异步处理'''

start = now()

url = "http://www.baidu.com/s?wd="

rets = []

for i in range(10):

# 使用tasks.urlopen.delay()的方法传入url参数, 此时urlopen函数就不会立刻被调用, 而是把这个task作为一个任务传递给celery的任务队列, 在任务队列中保存起来, 再由对应的worker从列队中取出任务进行执行.

ret = tasks.urlopen.delay(url + str(i))

rets.append(ret)

print("耗时：", now() - start)

def no\_celery():

'''如果直接调用函数，那么将不会交由celery处理, 相当于只是把urlopen原本的功能拿来使用了'''

start = now()

url = "http://www.baidu.com/s?wd="

for i in range(10):

tasks.urlopen(url+str(i))

print("耗时：", now() - start)

if \_\_name\_\_ == '\_\_main\_\_':

use\_celery()

# no\_celery()

worker.py：启动一个celery的worker工作进程

import os

# 启动一个celery的worker工作进程

# -A 指定要运行的任务模块

# -l 执行log日志等级

# -P gevent表明使用gevent来运行任务, 如果没有-P参数, 会以默认的线程方式进行执行

os.system("celery worker -A tasks -l info -P gevent")

程序启动之前, 先启动redis容器, 把代码中的redis参数修改为redis容器中的参数, 然后使用redis desktop manager连接到redis数据库中, 查看db0和db1中是否有数据.

然后启用spider.py, 向redis中的任务队列中添加任务. 查看redis中的db0, 其中就会出现已经添加的任务.

spider.py只是把任务添加到了redis的任务队列中, 想要真正执行任务, 还要启动worker. 可以直接运行worker.py, 也可以通过终端在tasks.py所在的目录中执行以下代码来运行worker

celery worker -A tasks -l info -P gevent

查看redis数据库中的db1, 在其中就会看到执行的结果, 是json格式的字符串. 查看tasks.py中的urlopen()函数, 其中return的内容必须是一个可以被json序列化的数据, 否则在保存时会出现问题.

### 4.6. 爬虫中异步库使用总结

**学习难度：**

* asyncio：3星
* gevent：2星
* tornado：3星
* twisted：5星
* celery：3星

**上手难度：**

* asyncio：3星
* gevent：1星
* tornado：2星
* twisted：5星
* celery：2星

**兼容程度：**

* asyncio：3星
* gevent：2星
* tornado：1星
* twisted：4星
* celery：5星