并发实现模块和相关包

一、concurrent.futures模块

这个小节主要讨论python3.2 引入的concurrent.futures模块。之前的版本使用这个模块需要手动安装 futures 这个模块。

这里会介绍future的概念。 future是一种对象，表示异步执行的操作。它是concurrent.futures模块和下一节asyncio包的基础。

Python标准库为我们提供了threading和multiprocessing模块编写相应的多线程/多进程代码。而concurrent.futures模块，它提供了ThreadPoolExecutor和ProcessPoolExecutor两个类， 实现了对threading和multiprocessing的更高级的抽象，对编写线程池/进程池提供了直接的支持。

> concurrent.futures模块，可以利用multiprocessing实现真正的平行计算。
>
> 核心原理是：concurrent.futures会以子进程的形式，平行的运行多个python解释器，从而令python程序可以利用多核CPU来提升执行速度。由于子进程与主解释器相分离，所以他们的全局解释器锁也是相互独立的。每个子进程都能够完整的使用一个CPU内核。

concurrent.futures基础模块是executor和submit。

先来看一段代码

```python
from concurrent import futures

def download_pic(url):
    image = get_image(url)  # 获取图片
    save(image)  # 存储图片
    return url


def down_pictures(pic_list):
    """
    pic_list: 表示图片地址列表
    """
    results = []
    with futures.ThreadPoolExecutor(max_workers=3) as executor:
        tasks = []
        for url in pic_list[:5]:
            future = executor.submit(download_pic, url)
            tasks.append(future)
            print(url, future)

        for future in futures.as_completed(tasks):
            res = future.result()
            print(future, res)
            results.append(res)

    return results
```

1.1 executor

Future可以理解为一个在未来完成的操作，这是异步编程的基础。通常情况下，我们执行io操作，如访问url时在等待结果返回之前会产生阻塞， cpu不能做其他事情，而Future的引入帮助我们在等待的这段时间可以完成其他的操作。

Executor是一个抽象类，它不能被直接使用。它为具体的异步执行定义了一些基本的方法。ThreadPoolExecutor和ProcessPoolExecutor继承了Executor。 值得一提的是Executor实现了*_enter_*和*_exit_*使得其对象可以使用with操作符。使得当任务执行完成之后，自动执行shutdown函数，而无需编写相关释放代码。

- ThreadPoolExecutor对象

ThreadPoolExecutor类是Executor子类，使用线程池执行异步调用.

class concurrent.futures.ThreadPoolExecutor(max_workers)

使用max_workers数目的线程池执行异步调用

- ProcessPoolExecutor对象

ThreadPoolExecutor类是Executor子类，使用进程池执行异步调用.

class concurrent.futures.ProcessPoolExecutor(max_workers=None)

使用 max*workers 数目的进程池执行异步调用，如果max*workers为None则使用机器的处理器数目（如4核机器max_worker配置为None时，则使用4个进程进行异步并发）

- submit()

Executor中定义了 submit(fn, *args, **kwargs)方法， fn执行的函数fn(*args, **kwargs)。这个方法的作用是提交一个可执行回调的任务， 然后返回一个Futurn对象，也就是给定的回调。

```python
# -*- coding:utf-8 -*-

from concurrent import futures

import requests


def request_data(url, timeout=10):
    response = requests.get(url, timeout=timeout)
    return url, response.status_code


with futures.ThreadPoolExecutor(max_workers=3) as executor:
    url = 'http://www.baidu.com'
    future = executor.submit(request_data, url)
    print(future.result())
```

如果要提交多个任务， 可以通过循环，多次submit()

- map()

Executor.map(func, *iterables, timeout=None)相当于map(func, *iterables)，但是func是异步执行。timeout的值可以是int或float，如果操作超时，会返回raisesTimeoutError；如果不指定timeout参数，则不设置超时间。

> - func：需要异步执行的函数
> - *iterables：可迭代对象，如列表等。每一次func执行，都会从iterables中取参数。
> - timeout：设置每次异步操作的超时时间

```python
# -*- coding:utf-8 -*-

from concurrent import futures

import requests


def request_data(url, timeout=10):
    response = requests.get(url, timeout=timeout)
    return url, response.status_code


with futures.ThreadPoolExecutor(max_workers=3) as executor:
    url_list = ['http://www.baidu.com', 'http://www.jd.com', 'http://sina.
    results = executor.map(request_data, url_list)
    for future in results:
        print(future)
```

1.2 Future

Future可以理解为一个在未来完成的操作，这是异步编程的基础。通常情况下，我们执行io操作，访问url时在等待结果返回之前会产生阻塞， cpu不能做其他事情，而Future的引入帮助我们在等待的这段时间可以完成其他的操作。

Executor.submit函数返回future对象，future提供了跟踪任务执行状态的方法。比如判断任务是否执行中future.running()，判断任务是否执行完成future.done()等等。

- as_completed

as_completed方法传入futures迭代器和timeout两个参数。

默认timeout=None，阻塞等待任务执行完成，并返回执行完成的future对象迭代器，迭代器是通过yield实现的。

timeout>0，等待timeout时间，如果timeout时间到仍有任务未能完成，不再执行并抛出异常TimeoutError

as_completed不是按照元素的顺序返回的。

```python
from concurrent import futures


def download_pic(url):
    image = get_image(url)  # 获取图片
    save(image)  # 存储图片
    return url


def down_pictures(pic_list):
    """
    pic_list: 表示图片地址列表
    """
    results = []
    with futures.ThreadPoolExecutor(max_workers=3) as executor:
        tasks = []
        for url in pic_list[:5]:
            future = executor.submit(download_pic, url)
            tasks.append(future)
            print(url, future)

        for future in futures.as_completed(tasks):
            res = future.result()
            print(future, res)
            results.append(res)
    return results
```

- wait

wait方法接会返回一个tuple(元组)，tuple中包含两个set(集合)，一个是completed(已完成的)另外一个是uncompleted(未完成的)。使用wait方法的一个优势就是获得更大的自由度，它接收三个参数FIRST*COMPLETED, FIRST*EXCEPTION和ALL*COMPLETE，默认设置为ALL*COMPLETED。

> - FIRST_COMPLETED - Return when any future finishes or is cancelled.
> - FIRST*EXCEPTION - Return when any future finishes by raising an exception. If no future raises an exception then it is equivalent to ALL*COMPLETED.
> - ALL_COMPLETED - Return when all futures finish or are cancelled.

```python

import time
from concurrent.futures import ProcessPoolExecutor, wait, ALL_COMPLETED, F


def gcd(pair):
    """
    计算最大公约数
    :param pair:
    :return:
    """
    a, b = pair
    low = min(a, b)
    for i in range(low, 0, -1):
        if a % i == 0 and b % i == 0:
            return i

if __name__ == '__main__':
    numbers = [
        (1963309, 2265973), (1879675, 2493670), (2030677, 3814172),
        (1551645, 2229620), (1988912, 4736670), (2198964, 7876293)
    ]

    start = time.time()
    with ProcessPoolExecutor(max_workers=2) as pool:
        futures = [ pool.submit(gcd, pair) for pair in numbers]
        for future in futures:
            print('执行中:%s, 已完成:%s' % (future.running(), future.done())
        print('#### 分界线 ####')
        done, unfinished = wait(futures, timeout=2, return_when=ALL_COMPLE
        for d in done:
            print('执行中:%s, 已完成:%s' % (d.running(), d.done()))
            print(d.result())
    end = time.time()
    print('Took %.3f seconds.' % (end - start))
```

总结

executor.map VS futures.as_completed&&executor.submit

(1) executor.map()函数易于使用，他的一个特点很独特：函数的返回结果的顺序和调用开始的顺序一致。具体使用可以根据实际情况来选定。 如果第一个调用生成结果耗时10s, 其他的调用耗时2s, 代码会阻塞10s, 获取map方法返回的生成器产出的第一个结果，后续的结果不会阻塞，因为后序的调用都完成。

如果我们处理的时候必须等到获取到所有结果才处理或者要求严格的顺序，操作没有问题。 而我们通

常的情况是，不管提交的顺序，只要获取到最后的结果就行，故结合submit()以及features.as_completed函数结合起来使用。

(2) Executor.submit和futures.as_completed这个组合比executor.map更灵活，因为submit能处理不同的可调用对象和参数，而executor.map只能 处理参数不同的同一个可调用对象。

此外futures.as_completed函数的future集合可以来自多个executor实例。例如一些ThreadPoolExecutor实例，一些 ProcessPoolExecutor实例。

二、sched

并发定时任务， 类似threading.Timer

```python
"""
def enter(self, delay, priority, action, argument=(), kwargs=_sentinel):
    # A variant that specifies the time as a relative time.

    # This is actually the more commonly used interface.
    time = self.timefunc() + delay
    return self.enterabs(time, priority, action, argument, kwargs)
"""

import sched, time
s = sched.scheduler(time.time, time.sleep)
def print_time(a='default'):
    print("From print_time", time.ctime(), a)

def print_some_times():
    print(time.ctime())
    s.enter(10, 1, print_time)
    s.enter(5, 2, print_time, argument=('positional',))
    s.enter(5, 1, print_time, kwargs={'a': 'keyword'})
    s.run()
    print(time.ctime())

print_some_times()
```

课后思考：sched和Timer的异同

三、Queue/queue

多线程做了说明，此处略过。

四、APScheduler

提供了基于日期、固定时间间隔以及crontab类型的任务，并且可以持久化任务。基于这些功能，我们可以很方便的实现一个python定时任务系统。

- 触发器:包含调度逻辑，每一个作业有它自己的触发器，用于决定接下来哪一个作业会运行。
- 作业存储: 存储被调度的作业，默认的作业存储是简单地把作业保存在内存中，其他的作业存储是将作业也可以保存在数据库中。
- 执行器: 处理作业的运行，他们通常通过在作业中提交指定的可调用对象到一个线程或者进城池来进行。当作业完成时，执行器将会通知调度器。
- 调度器: 调度器提供了处理处理作业存储、调度器和触发器这些的接口。配置作业存储和执行器可以在调度器中完成，例如添加、修改和移除作业。

```python
# 调度器对象
sched = BlockingScheduler()

# 添加作业
sched.add_job(my_job, 'interval', seconds=5)

#trigger 作业控制:

# 1. cron定时调度（某一定时时刻执行）
sched.add_job(my_job, 'cron', year=2018,month = 01,day = 30,hour = 18,minu
# https://www.cnblogs.com/longjshz/p/5779215.html

# 2 interval间隔多久
sched.add_job(my_job, 'interval',days  = 3,hours = 17,minutes = 19,seconds

# date
sched.add_job(my_job, 'date', run_date=date(2018, 01, 29), args=['text'])
```

```python
from datetime import datetime
import os
from time import sleep

from apscheduler.schedulers.blocking import BlockingScheduler


def tick():
    print('begin! The time is: %s' % datetime.now())
    sleep(6)
    print('end! The time is: %s' % datetime.now())


if __name__ == '__main__':
    scheduler = BlockingScheduler()
    # scheduler.add_job(tick, 'interval', seconds=3)
    scheduler.add_job(tick, 'cron', day='30', second='*/5')

    print('Press Ctrl+{0} to exit'.format('Break' if os.name == 'nt' else
    scheduler.start()
```


[1] crontab介绍使用(http://linuxtools-rst.readthedocs.io/zh_CN/latest/tool/crontab.html)

[2] APScheduler简单介绍(https://www.cnblogs.com/luxiaojun/p/6567132.html)

五、subprocess

子进程（subprocess包）

在python中，通过subprocess包，fork一个子进程，并运行外部程序。

调用系统的命令的时候，最先考虑的os模块。用os.system()和os.popen()来进行操作。但是这两个命令过于简单，不能完成一些复杂的操作，如给运行的命令提供输入或者读取命令的输出，判断该命令的运行状态，管理多个命令的并行等等。这时subprocess中的Popen命令就能有效的完成我们需要的操作。

该子模块允许你创建新的流程，连接到它们的输入/输出/错误管道，并获取他们的返回值。该模块打算替换多个旧的模块和功能：os.system和os.spawn。

使用subprocess时建议使用run()函数去处理所有它可以处理的情况，因为高级用法可以直接使用底层POPEN接口。

run()函数是Python 3.5中新添加的。 使用方法：

subprocess.run(args, *, stdin=None, input=None, stdout=None, stderr=None, shell=False,

timeout=None, check=False)

- args 该参数用于启动进程。这可能是一个列表或一个字符串。
- returncode 该参数表示子进程的退出状态。通常情况下，0作为退出状态表明它成功运行。 负值-N表明子进程被信号N终止（仅POSIX）

```
# 子进程输出当前目录下文件
>>> import subprocess
>>> subprocess.run(["ls"])
```

附录

with as语句

文件打开的演变

先看一下一般打开文件

```Python
file = open("/test.txt")
data = file.read()
file.close()
```

> - 1.忘记close
> - 2.数据异常后没有处理

完善一下，处理异常并关闭文件

```Python
file = open("/tmp/foo.txt")
try:
    data = file.read()
finally:
    file.close()
```

> 处理了异常等，但是代码比较冗长

with as 语法

```Python
with open("/tmp/foo.txt") as file:
    data = file.read()
```

除了有更优雅的语法，with还可以很好的处理上下文环境产生的异常

with如何工作?

上下文管理协议的对象:

- 对象内实现了两个方法：__enter__() 和__exit__()
- __enter__()方法会在with的代码\块执行之前执行，__exit__()会在代码块执行结束后执行。
- __exit__()方法内会自带当前对象的清理方法。

with as

紧跟with后面的语句被求值后，返回对象的__enter__()方法被调用，这个方法的返回值将被赋值给as后面的变量。当with后面的代码块全部被执行完之后，将调用前面返回对象的**exit**()方法。

with语句更简洁。而且更安全。代码量更少。

例子:

```python
with open(r'somefileName') as f:
        for line in f:
            print line
            # ...more code
```