concurrent.futures

fluent py 学习..

构造

- `concurrent.futures` 模块构造了ThreadPoolExecutor和ProcessPoolExecutor两个类，这两个类通过接口实现了在不同的线程或进程中执行可调用对象，并在内部维护着进程池或线程池，来对创造的线程或进程进行管理。

背后的future

- `concurrent.futures` 模块中构造了 `Future` 类，这个类的实例表示可能已经完成的或尚未完成的延迟计算， `future` 封装待完成的操作，即可放入队列。实际上对线程的调度和管理就是通过对队列中的 `future` 对象进行操作而完成的。

关于迭代器，可迭代对象和yield

- 只要一个对象实现了 `__next__` 方法，这个对象就是迭代器，可迭代对象是指实现了能返回迭代器的 `__iter__` 方法的对象。因此，可迭代对象本身并不是迭代器，而是在可迭代对象需要被迭代时会自动调用它的 `__iter__` 方法生成一个对应的迭代器来进行迭代. 也就是说，python从实现了 `__iter__` 方法的可迭代对象中获取实现了 `__next__` 方法的迭代器. 当python解释器需要迭代对象x时，会自动调用 `iter(x)` ，也就是说：

```
for i in x:
    pass
```

实际上进行了：

```
for i in iter(x):
    pass
```

也就是：

```
for i in x.__iter__():
```

```
    pass
```

- 但是当这个对象没有实现 `__iter__` 方法，但实现了 `__getitem__` 方法时，python会自动创造一个迭代器，尝试按顺序获取元素
- 生成器 `yield` 是一种控制流程的方式， `yield item` 这行代码会产出一个值，提供给 `next()` 的调用方；此时，会暂停执行生成器，让程序的流程回归到调用方，调用方继续工作，直到调用方需要另一个值再次对生成器调用 `next()` 时，程序流程回到生成器，调用方继续从生成器中拉取值. 生成器实现了 `__next__` 方法，因此生成器也是一种迭代器

```
import time
def create_generator():
    i = 0
    yield i
    print("生成器开始睡觉1s了！")
    time.sleep(1)
    print("睡醒了\n")
    i = 1
    yield i
    print("生成器开始睡觉1s了！")
    time.sleep(1)
    print("睡醒了\n")


for i in create_generator():
    print("第{}个产出".format(i))
    print(i is None)
```

```
第0个产出
False
生成器开始睡觉1s了！
睡醒了

第1个产出
False
生成器开始睡觉1s了！
睡醒了
```

- `create_generator` 是一个函数， `create_generator()` 是这个函数产生的生成器， `for` 循环会隐式在这个生成器上执行 `next` 方法，在第1次循环中，生成器 `yield` 产生0, `yield` 产出值后程序的流程回归到主循环中，运行 `print` ，到下一个循环时，继续调用 `next` 方法，生成器中运行剩下的部分，直到遇到下一个 `yield` ，产出值后再停止.

- for 循环不但会隐式执行 next 方法，而且会监听 StopIteration 错误，因此在上面的循环中， next 被第三次调用时，生成器运行第二个 yield i 后面的代码，生成器结束， raise StopIteration ， for 捕捉到这个错误，循环结束.

```
import time
from concurrent import futures


def download_flag(n):
    print(time.strftime("[%H:%M:%S]") + "开始了第{}个线程".format(n))
    time.sleep(5 - n)
    print(time.strftime("[%H:%M:%S]") + "结束了第{}个线程".format(n))
    return n


num_list = [1, 2, 3]


def batch_downloads():

    workers = 3

    with futures.ThreadPoolExecutor(workers) as executor:
        future_tasks = []
        for cc in num_list:
            future = executor.submit(download_flag, cc)
            future_tasks.append(future)

        task_iter = futures.as_completed(future_tasks)
        done_list = []
        for future in task_iter:
            print(time.strftime("[%H:%M:%S]") + " 在循环中！")
            done_list.append(future.result())
        return done_list


if __name__ == "__main__":
    print(batch_downloads())
```

```
[15:13:07]开始了第1个线程
[15:13:07]开始了第2个线程
[15:13:07]开始了第3个线程
[15:13:09]结束了第3个线程
[15:13:09] 在循环中！
[15:13:10]结束了第2个线程
[15:13:10] 在循环中！
[15:13:11]结束了第1个线程
```

```
[15:13:11] 在循环中!
[3, 2, 1]
```

future的一些接口：

- Executor(ThreadPoolExecutor or ProcessPoolExecutor)的 `submit` 方法接收一个可调用对象，将其提交到线程池，并返回一个 `future` 对象，通过这个 `future` 对象提供的接口可以查看这个可调用对象的状况，运行的结果等。
- executor对象的 `__exit__` 方法，会调用 `executor.shutdown(wait=True)` 方法，会在所有线程都执行完毕前阻塞线程。也就是说只有所有线程都执行完毕后，程序才会结束 `with` 中的内容向下继续进行，否则会一直停留在 `with` 中.
- `future` 对象具有 `done` 方法，指明 `future` 对应的对象是否已完成，该方法不阻塞.
- `future` 对象具有 `result` 方法， `result` 方法会阻塞调用方所在的进程，直到有结果时才会返回(这是因为 `result` 方法用 `return` 作返回值),如果存在多个线程，若第一个线程需要运行10s，而剩下的各个线程只需要运行1s时，这时若想要返回结果时，若先调用了第一个线程的 `result` 方法，此时主进程会阻塞10s才能获得所有线程的返回结果，而在第2s的时候剩下的线程就已经运行完毕.(阻塞是阻塞调用 `result` 方法的进程向下进行，而不是阻塞子线程的运行)
- `concurrent.futures` 中的 `as_completed` 函数解决了这个问题，这个函数的参数是一个 `future` 列表，返回的结果是一个生成器，一旦 `future` 列表中有 `future` 对象运行完毕了，这个生成器就会把这个 `future` 对象 `yield` 出来，返回结果是按照执行完毕的顺序进行的.所以在上面的程序中运行 `done_list.append(future.result())` 也是不会阻塞的,因为此时获得的 `future` 就是最先运行完的 `future` 对象.
- `as_completed` 函数会打乱原有提交的顺序，因此可以提前构建一个字典，把各个 `future` 映射到其他数据，这样，尽管 `as_completed` 收集到的结果顺序已经乱了，但是依然可以方便用于后续处理

关于executor.map方法

```
"""
Experiment with ``ThreadPoolExecutor.map``
"""
# BEGIN EXECUTOR_MAP
from time import sleep, strftime
from concurrent import futures


def display(*args):  # <1>
```

```
        print(strftime('[%H:%M:%S]'), end=' ')
        print(*args)


    def loiter(n):  # <2>
        msg = '{}loiter({}): doing nothing for {}s...'
        display(msg.format('\t'*n, n, n))
        sleep(n)
        msg = '{}loiter({}): done.'
        display(msg.format('\t'*n, n))
        return n * 10  # <3>


    def main():
        display('Script starting.')
        executor = futures.ThreadPoolExecutor(max_workers=3)  # <4>
        results = executor.map(loiter, range(5))  # <5>
        display('results:', results)  # <6>.
        display('Waiting for individual results:')
        for i, result in enumerate(results):  # <7>
            display('result {}: {}'.format(i, result))


    main()
    # END EXECUTOR_MAP
```

```
[15:13:11] Script starting.
[15:13:11] loiter(0): doing nothing for 0s...
[15:13:11] loiter(0): done.
[15:13:11]     loiter(1): doing nothing for 1s...
[15:13:11]             loiter(2): doing nothing for 2s...
[15:13:11]                     loiter(3): doing nothing for 3s...
[15:13:11] results: <generator object Executor.map.<locals>.result_iterator at 0x000002E14
[15:13:11] Waiting for individual results:
[15:13:11] result 0: 0
[15:13:12]     loiter(1): done.
[15:13:12]                             loiter(4): doing nothing for 4s...
[15:13:12] result 1: 10
[15:13:13]             loiter(2): done.
[15:13:13] result 2: 20
[15:13:14]                     loiter(3): done.
[15:13:14] result 3: 30
[15:13:16]                             loiter(4): done.
[15:13:16] result 4: 40
```

- `executor.map` 实现了同时并发或并行(取决于executor的种类)的功能，方法的第一个参数是要调用的函数，后面是函数的参数列表，若列表中有n个元素，则会并发提交n个任务给

executor，这是非阻塞调用(因为此时只是提交，并不涉及返回结果)。同时提交n个任务不意味着会同时并发运行n个任务，同时并发运行的任务数量和executor的线程数是一致的，在这里是3，因此同时只能进行三个线程，因此如上输出结果所示，一开始只有loiter(0),loiter(1)和loiter(2)开始了. 而loiter(0)瞬间就运行结束了，因此loiter(0)的这个线程马上就开始运行loiter(3)，等loiter(1)运行好了，他开始运行loiter(4)...

- `executor.map` 方法会返回一个生成器，但是这个生成器yield的不是优先完成的future的result，而是按照map顺序的result，因此如果第一个调用会耗时很长，这里迭代获取结果时会发生阻塞. 和 `as_completed` 并不一致.

```
if __name__ == "__main__":
    __spec__ = "ModuleSpec(name='builtins', loader=<class '_frozen_importlib.BuiltinImp
    with futures.ProcessPoolExecutor(8) as executor:
        datasets = list(
            executor.map(generate_feture, np.array_split(range(1, FILE_NUMBER), 8))
        )
    all_feature = np.concatenate([dataset[0] for dataset in datasets])
    all_target = np.concatenate([dataset[1] for dataset in datasets])
    np.save("feature.npy", all_feature)
    np.save("target.npy", all_target)
```

- 最近写了一个简单的8核同时处理数据的并行，文件编号从1到FILE_NUMBER，这时 `np.array_split` 函数很有用，可以把一个可迭代对象均匀分割为n份，如果不能均匀，也会自动舍入，保持近似均分，对于并行很有用
- 一定要注意到 `executor.map` 方法的返回是一个生成器，迭代一次就没了.可以先将其转换为列表. 如果只需要迭代一次，则不需要转换，这时也会大大节约内存. 内存占用过大的列表都可以考虑用迭代器去替代.
- .npy文件比.txt文件快得多
- 由于Windows没有fork，多处理模块启动一个新的Python进程并导入调用模块。如果没有 `if __name__ == '__main __'` ，这将启动无限继承的新进程（或直到机器耗尽资源）
- jupyter本身也只是一个python进程，jupyter只能跟踪主进程，没法跟踪子进程。因此想要用通过concurrent或multiprocessing运行多进程必须在.py文件中运行