concurrent包

concurrent.futures

3.2版本引入的模块。

异步并行任务编程模块，提供一个高级的异步可执行的便利接口。

提供了2个池执行器
ThreadPoolExecutor 异步调用的线程池的Executor
ProcessPoolExecutor 异步调用的进程池的Executor

ThreadPoolExecutor对象

首先需要定义一个池的执行器对象，Executor类子类对象。

方法	含义
ThreadPoolExecutor(max_workers=1)	池中至多创建max_workers个线程的池来同时异步执行，返回Executor实例
`submit(fn, *args, **kwargs)`	提交执行的函数及其参数，返回Future实例
shutdown(wait=True)	清理池

Future类

方法	含义
done()	如果调用被成功的取消或者执行完成，返回True
cancelled()	如果调用被成功的取消，返回True
running()	如果正在运行且不能被取消，返回True
cancel()	尝试取消调用。如果已经执行且不能取消返回False，否则返回True
result(timeout=None)	取返回的结果，timeout为None，一直等待返回；timeout设置到期，抛出concurrent.futures.TimeoutError 异常
	取返回的异常，timeout为None，一直等待返回；timeout设置到

exception(timeout=None)	期，抛出concurrent.futures.TimeoutError 异常

```
# ThreadPoolExecutor例子
import threading
from concurrent import futures

import logging
import time

# 输出格式定义
FORMAT = '%(asctime)-15s\t [%(processName)s:%(threadName)s, %(process)d:%(thread)8d
] %(message)s'
logging.basicConfig(level=logging.INFO, format=FORMAT)

def worker(n):
    logging.info('begin to work{}'.format(n))
    time.sleep(5)
    logging.info('finished{}'.format(n))

# 创建线程池，池的容量为3
executor = futures.ThreadPoolExecutor(max_workers=3)
fs = []
for i in range(3):
    future = executor.submit(worker, i)
    fs.append(future)

for i in range(3, 6):
    future = executor.submit(worker, i)
    fs.append(future)

while True:
    time.sleep(2)
    logging.info(threading.enumerate())

    flag = True
    for f in fs: # 判断是否还有未完成的任务
        logging.info(f.done())
        flag = flag and f.done()
        # if not flag: # 注释if看的清楚
        #     break
```

```
    print('-'*30)

    if flag:
        executor.shutdown() # 清理池，池中线程全部杀掉
        logging.info(threading.enumerate())
        break

# 线程池一旦创建了线程，就不需要频繁清除
```

ProcessPoolExecutor对象

方法一样。就是使用多进程完成。

```
import threading
from concurrent import futures

import logging
import time

# 输出格式定义
FORMAT = '%(asctime)-15s\t [%(processName)s:%(threadName)s, %(process)d:%(thread)8d
] %(message)s'
logging.basicConfig(level=logging.INFO, format=FORMAT)

def worker(n):
    logging.info('begin to work{}'.format(n))
    time.sleep(5)
    logging.info('finished{}'.format(n))

if __name__ == '__main__':
    # 创建进程池，池的容量为3
    executor = futures.ProcessPoolExecutor(max_workers=3)
    fs = []
    for i in range(3):
        future = executor.submit(worker, i)
        fs.append(future)

    for i in range(3, 6):
```

```
        future = executor.submit(worker, i)
        fs.append(future)

    while True:
        time.sleep(2)
        logging.info(threading.enumerate())

        flag = True
        for f in fs: # 判断是否还有未完成的任务
            logging.info(f.done())
            flag = flag and f.done()
            # if not flag: # 注释if看的清楚
            #     break

        print('-'*30)

        if flag:
            executor.shutdown()  # 清理池。除非不用，不用频繁清理池
            logging.info(threading.enumerate()) # 多进程时看主线程已没有必要了
            break
```

支持上下文管理

concurrent.futures.ProcessPoolExecutor继承自concurrent.futures.*base.Executor*，而父类有__enter__、__exit__方法，支持上下文管理。可以使用with语句。
__exit__方法本质还是调用的shutdown(wait=True)，就是一直阻塞到所有运行的任务完成

使用方法

```
with ThreadPoolExecutor(max_workers=1) as executor:
    future = executor.submit(pow, 323, 1235)
    print(future.result())
```

使用上下文改造上面的例子，增加返回计算的结果

```
import threading
from concurrent import futures

import logging
```

```
import time

# 输出格式定义
FORMAT = '%(asctime)-15s\t [%(processName)s:%(threadName)s, %(process)d:%(thread)8d
] %(message)s'
logging.basicConfig(level=logging.INFO, format=FORMAT)

def worker(n):
    logging.info('begin to work{}'.format(n))
    time.sleep(5)
    logging.info('finished{}'.format(n))
    return n + 100 # 返回结果

if __name__ == '__main__':
    # 创建进程池，池的容量为3
    executor = futures.ProcessPoolExecutor(max_workers=3)

    with executor: # 上下文
        fs = []
        for i in range(3):
            future = executor.submit(worker, i)
            fs.append(future)

        for i in range(3, 6):
            future = executor.submit(worker, i)
            fs.append(future)

        while True:
            time.sleep(2)
            logging.info(threading.enumerate())

            flag = True
            for f in fs: # 判断是否还有未完成的任务
                logging.info(f.done())
                flag = flag and f.done()
                if f.done(): # 如果做完了，看看结果
                    logging.info("result = {}".format(f.result()))
                # if not flag: # 注释if看的清楚
                #     break
```

```
            print('-'*30)
            if flag:  break
    # executor.shutdown()  # 上下文清理了资源
    logging.info('====end=====')
    logging.info(threading.enumerate())  # 多进程时看主线程已没有必要了
```

总结

该库统一了线程池、进程池调用，简化了编程。
是Python简单的思想哲学的体现。

唯一的缺点：无法设置线程名称。但这都不值一提。