多进程

由于Python的GIL全局解释器锁存在，多线程未必是CPU密集型程序的好的选择。
多进程可以完全独立的进程环境中运行程序，可以较充分地利用多处理器。
但是进程本身的隔离带来的数据不共享也是一个问题。而且线程比进程轻量级。

multiprocessing

Process类

Process类遵循了Thread类的API，减少了学习难度。

先看一个例子，前面介绍的单线程、多线程比较的例子的多进程版本

```
import multiprocessing
import datetime

def calc(i):
    sum = 0
    for _ in range(1000000000): # 10亿
        sum += 1
    return i, sum

if __name__ == '__main__':
    start = datetime.datetime.now() # 注意一定要有这一句

    ps = []
    for i in range(4):
        p = multiprocessing.Process(target=calc, args=(i,), name='calc-
{}'.format(i))
        ps.append(p)
        p.start()

    for p in ps:
        p.join()
        print(p.name, p.exitcode)

    delta = (datetime.datetime.now() - start).total_seconds()
    print(delta)
    for p in ps:
        print(p.name, p.exitcode)
    print('===end====')
```

对于上面这个程序，在同一主机（授课主机）上运行时长的对比

- 使用单线程、多线程跑了4分钟多
- 多进程用了1分半

看到了多个进程都在使用CPU，这是真并行，而且进程库几乎没有什么学习难度

注意：多进程代码一定要放在`__name__ == "__main__"`下面执行。

名称	说明
pid	进程id
exitcode	进程的退出状态码
terminate()	终止指定的进程

进程间同步

Python在进程间同步提供了和线程同步一样的类，使用的方法一样，使用的效果也类似。
不过，进程间代价要高于线程间，而且系统底层实现是不同的，只不过Python屏蔽了这些不同之处，让用户简单使用多进程。

multiprocessing还提供共享内存、服务器进程来共享数据，还提供了用于进程间通讯的Queue队列、Pipe管道。

通信方式不同

1. 多进程就是启动多个解释器进程，进程间通信必须序列化、反序列化
2. 数据的线程安全性问题
   如果每个进程中没有实现多线程，GIL可以说没什么用了

多进程、多线程的选择

1、CPU密集型
CPython中使用到了GIL，多线程的时候锁相互竞争，且多核优势不能发挥，选用Python多进程效率更高。
2、IO密集型
在Python中适合是用多线程，可以减少多进程间IO的序列化开销。且在IO等待的时候，切换到其他线程继续执行，效率不错。

应用

请求/应答模型：WEB应用中常见的处理模型

master启动多个worker工作进程，一般和CPU数目相同。发挥多核优势。
worker工作进程中，往往需要操作网络IO和磁盘IO，启动多线程，提高并发处理能力。worker处理用户的请求，往往需要等待数据，处理完请求还要通过网络IO返回响应。
这就是nginx工作模式。

concurrent.futures包

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类的子类实例。

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 支持上下文，进入时返回自己，退出时调用shutdown(wait=True)
submit(fn, *args, **kwargs)	提交执行的函数及其参数，如有空闲开启daemon线程，返回Future类的实例
shutdown(wait=True)	清理池，wait表示是否等待到任务线程完成

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
exception(timeout=None)	取返回的异常，timeout为None，一直等待返回；timeout设置到期，抛出concurrent.futures.TimeoutError 异常

```
from concurrent.futures import ThreadPoolExecutor, wait
import datetime
import logging

FORMAT = "%(asctime)s [%(processName)s %(threadName)s] %(message)s"
logging.basicConfig(format=FORMAT, level=logging.INFO)

def calc(base):
    sum = base
    for i in range(100000000):
        sum += 1
    logging.info(sum)
    return sum

start = datetime.datetime.now()
executor = ThreadPoolExecutor(3)
with executor: # 默认shutdown阻塞
    fs = []
    for i in range(3):
        future = executor.submit(calc, i*100)
        fs.append(future)
```

```
    #wait(fs) # 阻塞
    print('-' * 30)
for f in fs:
    print(f, f.done(), f.result()) # done不阻塞，result阻塞
print('=' * 30)

delta = (datetime.datetime.now() - start).total_seconds()
print(delta)
```

ProcessPoolExecutor对象

方法一样。就是使用多进程完成。

```
from concurrent.futures import ProcessPoolExecutor, wait
import datetime
import logging

FORMAT = "%(asctime)s [%(processName)s %(threadName)s] %(message)s"
logging.basicConfig(format=FORMAT, level=logging.INFO)

def calc(base):
    sum = base
    for i in range(100000000):
        sum += 1
    logging.info(sum)
    return sum

if __name__ == '__main__':
    start = datetime.datetime.now()
    executor = ProcessPoolExecutor(3)
    with executor: # 默认shutdown阻塞
        fs = []
        for i in range(3):
            future = executor.submit(calc, i*100)
            fs.append(future)

        #wait(fs) # 阻塞
        print('-' * 30)
    for f in fs:
        print(f, f.done(), f.result()) # done不阻塞，result阻塞
    print('=' * 30)

    delta = (datetime.datetime.now() - start).total_seconds()
    print(delta)
```

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。

马哥教育
IT人的高薪职业学院