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)
    print('===end====')
```

对于上面这个程序，在使用单线程、多线程都跑了4分钟多，而多进程用了1分半，这是真并行。

可以看出，几乎没有什么学习难度

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

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

进程池举例

multiprocessing.Pool 是进程池类。

名称	说明
apply(self, func, args=(), kwds={})	阻塞执行，导致主进程执行其他子进程就像一个个执行
apply_async(self, func, args=(), kwds={}, callback=None, error_callback=None)	与apply方法用法一致，非阻塞异步执行，得到结果后会执行**回调**
close()	关闭池，池不能再接受新的任务
terminate()	结束工作进程，不再处理未处理的任务
join()	主进程阻塞等待子进程的退出， join方法要在close或terminate之后使用

```
# 同步调用
import logging
import datetime
import multiprocessing

# 日志打印进程id、进程名、线程id、线程名
logging.basicConfig(level=logging.INFO, format="%(process)d %(processName)s %(thread)d %
(message)s")

def calc(i):
    sum = 0
```

```
    for _ in range(1000): # 10亿
        sum += 1
    logging.info(sum)
    return i, sum # 进程要return，才可以拿到这个结果

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

    pool = multiprocessing.Pool(4)
    for i in range(4):
        # 返回值，同步调用，注意观察CPU使用
        ret = pool.apply(calc, args=(i,))
        print(ret)
    pool.close()
    pool.join()

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

```
# 异步调用
import logging
import datetime
import multiprocessing

# 日志打印进程id、进程名、线程id、线程名
logging.basicConfig(level=logging.INFO, format="%(process)d %(processName)s %(thread)d %
(message)s")

def calc(i):
    sum = 0
    for _ in range(1000000000): # 10亿
        sum += 1
    logging.info(sum)
    return i, sum # 进程要return，callback才可以拿到这个结果

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

    pool = multiprocessing.Pool(4)
    for i in range(4):
        # 异步拿到的返回值是什么?
        ret = pool.apply_async(calc, args=(i,))
        print(ret, '~~~~~~~') # 异步，如何拿到真正的结果呢?
    pool.close()
    pool.join()

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

```python
# 异步调用，拿最终结果
import logging
import datetime
import multiprocessing

# 日志打印进程id、进程名、线程id、线程名
logging.basicConfig(level=logging.INFO, format="%(process)d %(processName)s %(thread)d %
(message)s")

def calc(i):
    sum = 0
    for _ in range(1000000000): # 10亿
        sum += 1
    logging.info(sum)
    return i, sum # 进程要return，callback才可以拿到这个结果

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

    pool = multiprocessing.Pool(4)
    for i in range(4):
        # 异步拿到的返回值是什么？回调起了什么作用？
        ret = pool.apply_async(calc, args=(i,),
                callback=lambda ret: logging.info('{} in callback'.format(ret)))
        print(ret, '~~~~~~~')
    pool.close()
    pool.join()

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

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，Python多进程效率更高。
2、IO密集型
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。

Linux的特殊进程

在Linux/Unix中，通过父进程创建子进程。

僵尸进程

一个进程使用了fork创建了子进程，如果子进程终止进入僵死状态，而父进程并没有调用wait或者waitpid获取子进程的状态信息，那么子进程仍留下一个数据结构保存在系统中，这种进程称为僵尸进程。
僵尸进程会占用一定的内存空间，还占用了进程号，所以一定要避免大量的僵尸进程产生。有很多方法可以避免僵尸进程。

孤儿进程

父进程退出，而它的子进程仍在运行，那么这些子进程就会成为孤儿进程。孤儿进程会被init进程（进程号为1）收养，并由init进程对它们完成状态收集工作。
init进程会循环调用wait这些孤儿进程，所以，孤儿进程没有什么危害。

守护进程

它是运行在后台的一种特殊进程。它独立于控制终端并周期性执行某种任务或等待处理某些事件。
守护进程的父进程是init进程，因为其父进程已经故意被终止掉了。
守护进程相对于普通的孤儿进程需要做一些特殊处理。