多线程

线程

进程 是计算机中的程序关于某数据集合上的一次运行活动，是系统进行资源分配和调度的基本单位。

进程是正在进行的一个过程或者说一个任务，而负责执行任务的则是CPU，进程本身是 一个抽象的概念,即进程就是一个过程、一个任务。

线程 是进程的一个实体，是CPU调度和分派的基本单位，它是比进程更小的能独立运行的基本单位。线程自己基本上不拥有系统资源，只拥有一点在运行中必不可少的资源(如程序计数器,一组寄存器和栈)，但是它可与同属一个进程的其他的线程共享进程所拥有的全部资源。

1. 全局解释器锁GIL

首先需要明确的一点是GIL并不是Python的特性，它是在实现Python解析器(CPython)时所引入的一个概念。

Python代码的执行由Python 虚拟机(也叫解释器主循环，CPython版本)来控制，Python 在设计之初就考虑到要在解释器的主循环中，同时只有一个线程在执行，即在任意时刻，只有一个线程在解释器中运行。对Python 虚拟机的访问由全局解释器锁（GIL）来控制，正是这个锁能保证同一时刻只有一个线程在运行。这是因为大部分解释程序代码和第三方C代码都不是线程安全的，需要保护。

对Python虚拟机的访问是由全局解释器锁(GIL)控制的。这个锁就是用来保证同时只能有一个线程运行的。

```
在多线程环境中,Python虚拟机将按照下面所述的方式执行
1.设置 GIL。
2.切换进一个线程去运行。
3.执行下面操作之一:
    a.指定数量的字节码指令。
    b.线程主动让出控制权(可以调用 time.sleep(0)来完成)。
4.把线程设置回睡眠状态(切换出线程)。
5.解锁 GIL。
6.重复上述步骤。
```

> **全局解释器锁GIL设计理念与限制**
>
> 1. GIL的设计简化了CPython的实现。锁住全局解释器使得比较容易的实现对多线程的支持，

> 但也损失了多处理器主机的并行计算能力。
>
> 2. 不论标准的，还是第三方的扩展模块，都被设计成在进行密集计算任务时，释放GIL。
> 3. 在做I/O操作时，GIL总是会被释放。对所有面向I/O 的(会调用内建的操作系统C 代码的)程序来说，GIL 会在这个I/O 调用之前被释放，以允许其它的线程在这个线程等待I/O 的时候运行。如果是纯计算的程序，没有 I/O 操作，解释器会每隔 100 次操作就释放这把锁，让别的线程有机会执行，如果某线程并未使用很多I/O 操作，它会在自己的时间片内一直占用处理器（和GIL）。也就是说，I/O 密集型的Python 程序比计算密集型的程序更能充分利用多线程环境的好处。

多线程

Python提供了多个模块来支持多线程编程，包括thread(2.x)、threading和Queue/queue模块等。可以使用thread和threading模块来创建与管理线程。

thread模块提供了基本的线程和锁定支持

而threading模块提供了更高级别、功能更全面的线程管理。

使用Queue模块,用户可以创建一个队列数据结构,用于在多线程之间进行共享。

我们将分别来查看这几个模块, 并给出几个例子和中等规模的应用。

1. thread模块(python2.x)

函数式：调用thread模块中的start*new*thread()函数来产生新线程。语法如下:

thread.start*new*thread (function, args[, kwargs]) 参数说明:

function - 线程函数。 args - 传递给线程函数的参数,他必须是个tuple类型。 kwargs - 可选参数。

```python
# 代码1.1
import thread
import time

# 为线程定义一个函数
def print_time(threadName, delay):
    """
    threadName 线程, delay 延迟时间
    """
    count = 0
    for i in range(4):
        time.sleep(delay)
        print "%s: %s" % (threadName, time.ctime(time.time()))


# 创建两个线程
try:
    thread.start_new_thread(print_time, ("线程-1", 2,))
    thread.start_new_thread(print_time, ("线程-2", 4,))
except:
    print "Error: unable to start thread"
```

它对于进程何时退出没有控制, thread 模块拥有的同步原语很少

推荐 使用更高级别的 threading 模块,而不使用 thread 模块。

2. threading模块

下面是threading 模块里所有的对象：

threading模块对象	描述
Thread	表示一个线程的执行的对象
Lock	锁原语对象（跟thread模块里的锁对象相同）
RLock	可重入锁对象。使单线程可以再次获得已经获得了的锁（递归锁定）
Condition	条件变量对象能让一个线程停下来，等待其他线程满足了某个“条件”。如状态的改变或值的改变
Event	通用的条件变量。多个线程可以等待某个时间的发生，在事件发生后，所有的线程都被激活
Semaphore	为等待锁的线程提供一个类似“等候室”的结构
BoundedSemaphore	与Semaphore类似，只是它不允许超过初始值
Timer	与thread类似，只是它要等待一段时间后才开始运行

2.1 Thread 类

threading的Thread类是主要的运行对象。它有很多thread模块里没有的函数。

函数	描述
start()	开始线程的执行
run()	定义线程的功能的函数（一般会被子类重写）
join(timeout=None)	程序挂起，直到线程结束；如果给了timeout，则最多阻塞timeout秒
getName()	返回线程的名字
setName(name)	设置线程的名字
isAlive()	布尔标志，表示这个线程是否还在运行中
isDaemon()	返回线程的daemon标志
setDaemon(daemonic)	把线程的daemon标志设为daemonic（一定要在调用start()函数前调用）

守护线程

另一个避免使用 thread 模块的原因是，它不支持守护线程。当主线程退出时，所有的子线程不论它们是否还在工作，都会被强行退出。有时，我们并不期望这种行为，这时，就引入了守护线程的概念

threading 模块支持守护线程，它们是这样工作的:

守护线程一般是一个等待客户请求的服务器，如果没有客户出请求，它就在那等着。如果你设定一个线程为守护线程，就表示你在说这个线程是不重要的，在进程退出的时候，不用等待这个线程退出。

可以通过daemon属性来设置守护线程。

serDeamon(False)(默认)前台线程，主线程执行过程中，前台线程也在进行，主线程执行完毕后，等待前台线程也执行完成后，主线程停止

Python

```python
# 代码1.2
# coding:utf-8

import threading
import time

def action(arg):
    time.sleep(1)
    print('sub thread start!the thread name is:%s\r' % threading.currentTh
    print('the arg is:%s\r' %arg)
    time.sleep(1)

for i in range(4):
    t =threading.Thread(target=action,args=(i,))
    t.setDaemon(False)#设置线程为后台线程
    t.start()

print('main_thread end!')
```

2.2 Lock & RLock

原语锁定是一个同步原语，状态是锁定或未锁定。两个方法acquire()和release() 用于加锁和释放锁。

由于线程之间随机调度：某线程可能在执行n条后，CPU接着执行其他线程。为了多个线程同时操作一个内存中的资源时不产生混乱，我们使用锁。

Lock（指令锁）是可用的最低级的同步指令。Lock处于锁定状态时，不被特定的线程拥有。 可以认为Lock有一个锁定池，当线程请求锁定时，将线程至于池中，直到获得锁定后出池。池中的线程处于状态图中的同步阻塞状态。

RLock（可重入锁）是一个可以被同一个线程请求多次的同步指令。RLock使用了“拥有的线程”和“递归等级”的概念，处于锁定状态时，RLock被某个线程拥有。拥有RLock的线程可以再次调用acquire()，释放锁时需要调用release()相同次数

简言之：Lock属于全局，Rlock属于线程

2.3 Condition

Condition（条件变量）和 Lock 参数一样，也是一个，也是一个同步原语.通常与一个锁关联。需要在多个Contidion中共享一个锁时，可以传递一个Lock/RLock实例给构造方法，否则它将自己生成一个RLock实例。

可以认为，除了Lock带有的锁定池外，Condition还包含一个等待池，池中的线程处于等待阻塞状态，直到另一个线程调用notify()/notifyAll()通知；得到通知后线程进入锁定池等待锁定。

2.4 Event

事件用于在线程间通信。一个线程发出一个信号，其他一个或多个线程等待。Event 通过通过内部标记来协调多线程运 。 方法 wait() 阻塞线程执 ，直到标记为 True。 set() 将标记设为 True，clear() 更改标记为 False。isSet() 用于判断标记状态。

2.5 timer

Timer（定时器）是Thread的派生类，用于在指定时间后调用一个方法。

class threading.Timer(interval, function, args=[], kwargs={}) 创建一个timer，在interval秒过去之后，它将以参数args和关键字参数kwargs运行function 。

```python
# 代码1.3
# coding: utf-8
from threading import Timer
import time


def fun():
    print("hello, world")


if __name__ == '__main__':
    print(time.time())
    t = Timer(5.0, fun)
    t.start()  # 5秒后, "hello, world"将被打印
    print(time.time())
```

2.6 local类

用于管理 thread-local（线程局部的）数据。对于同一个local，线程无法访问其他线程设置的属性；线程设置的属性不会被其他线程设置的同名属性替换。

可以把local看成是一个“线程-属性字典”的字典，local封装了从自身使用线程作为 key检索对应的属性字典、再使用属性名作为key检索属性值的细节。

threading 实例代码

```python
# 代碼1.4
from time import ctime
import threading

def coding(language):
    for i in range(5):
        print('I\'m coding ',language, ' program at ', ctime() )

def music():
    for i in range(5):
        print('I\'m listening music at ', ctime())

if __name__ == '__main__':

    print('thread %s is running...' % threading.current_thread().name)

    thread_list = []
    t1 = threading.Thread(target=coding, args=('Python',))
    t2 = threading.Thread(target=music)
    thread_list.append(t1)
    thread_list.append(t2)

    for t in thread_list:
        t.setDaemon(True)  # 设置为守护线程
        t.start()
        t.join()  # 在这个子线程完成运行之前，主线程将一直被阻塞

    print('thread %s ended.' % threading.current_thread().name)
```

3、Queue/queue

Queue用于建立和操作队列，常和threading类一起用来建立一个简单的线程队列。

队列有很多种，根据进出顺序来分类，可以分成

> - Queue.Queue(maxsize)　　FIFO（先进先出队列）
> - Queue.LifoQueue(maxsize)　　FIFO（先进先出队列）
> - Queue.PriorityQueue(maxsize)　　为优先度越低的越先出来

注意： 如果设置的maxsize小于1，则表示队列的长度无限长

FIFO是常用的队列，其一些常用的方法有：

> - Queue.qsize()　　返回队列大小
> - Queue.empty()　　判断队列是否为空

- Queue.full()　　判断队列是否满了
- Queue.get([block[,timeout]])　　从队列头删除并返回一个item，block默认为True，表示当队列为空却去get的时候会阻塞线程，等待直到有有item出现为止来get出这个item。如果是False的话表明当队列为空你却去get的时候，会引发异常。在block为True的情况下可以再设置timeout参数。表示当队列为空，get阻塞timeout指定的秒数之后还没有get到的话就引发Full异常。
- Queue.put(…[,block[,timeout]])　　向队尾插入一个item，同样若block=True的话队列满时就阻塞等待有空位出来再put，block=False时引发异常。同get的timeout，put的timeout是在block为True的时候进行超时设置的参数。
- Queue.task*done()*　　*从场景上来说，处理完一个get出来的item之后，调用task*done将向队列发出一个信号，表示本任务已经完成
- Queue.join() 监视所有item并阻塞主线程，直到所有item都调用了task_done之后主线程才继续向下执行

python2.x中，模块名为Queue python3.x中，模块名为queue

模块生产者消费者的场景是：生产者生产货物，然后把货物放到一个队列之类的数据结构中，生产货物所要花费的时间无法预先确定。消费者消耗生产者生产的货物的时间也是不确定的

```python
# 代码1.5
import queue,threading,time

q=queue.Queue()


def product(arg):
    while True:
        time.sleep(1)
        q.put(str(arg) + '包子')
        print('生产线程 %s 生产包子, 现有%s 个' % (str(arg), q.qsize()))


def consumer(arg):
    while True:
        q.get()
        print('消费线程 %s 消费包子, 现有%s 个' % (str(arg), q.qsize()))
        time.sleep(2)
for i in range(3):
    t=threading.Thread(target=product,args=(i,))
    t.start()
for j in range(2):
    t=threading.Thread(target=consumer,args=(j,))
    t.start()
```

线程池

线程池是预先创建线程的一种技术。线程池在还没有任务到来之前，创建一定数量的线程，放入空闲队列中。这些线程都是处于睡眠状态，即均为启动，不消耗 CPU，而只是占用较小的内存空间。当请求到来之后，缓冲池给这次请求分配一个空闲线程，把请求传入此线程中运行，进行处理。当预先创建的线程都处于运行 状态，即预制线程不够，线程池可以自由创建一定数量的新线程，用于处理更多的请求。当系统比较闲的时候，也可以通过移除一部分一直处于停用状态的线程。

concurrent.futures中ThreadPoolExecutor很好的实现了线程池。具体我们concurrent.futures内容中讲解。

多进程

python中的多线程其实并不是真正的多线程，CPU在同一时刻只能处理一个任务，只是因为cpu执行速度很快，cpu在各个任务之间来回的进行切换。

即如果想要充分地使用多核CPU的资源，在python中大部分情况需要使用多进程。

多进程中，每个进程中所有数据（包括全局变量）都各有拥有一份，互不影响。

一、进程

进程 是一个具有独立功能的程序关于某个数据集合的一次运行活动。是系统进行资源分配和调度的基本单位，它可以申请和拥有系统资源。

进程是CPU正在进行的一个过程或者说一个任务。

1. 创建进程

1.1 fork()函数

Unix/Linux/Mac操作系统都可以使用fork()函数来创建子进程（windows操作系统下没有此函数），分别在父进程和子进程内返回。

这是python中实现进程最底层的方法，其他方法从根本上也是利用fork()方法来实现的

```python
# 代码2.1
import os  #  导入os模块

# os.getpid()返回的是进程的id不是线程
print ('当前进程的ID是：%s' % os.getpid())

# 创建子进程，并返回进程的id，父进程返回的是父进程的id，子进程返回的是0
ID = os.fork()

if ID == 0:
    print ('这是子进程，ID是：%s。。父进程ID是：%s'
    % (os.getpid(), os.getppid()))
else:
    print ('这是父进程，ID是：%s' % os.getpid())
```

上面是操作系统层面创建进程的一个示例，实际中基本很少用。

1.2 multiprocessing.Process

python是跨平台的，所以自然肯定会为我们提供实现多进程的库，毕竟在win里面用不了fork()。此方法需要导入对应模块

```python
from multiprocessing import Process
p1=Process(target=func)
p1.start()
```

这个方法常用场景是使用少量进程做主动服务，如qq客户端，等这样的可以开多个。

Process类

创建进程的类：Process(group=None, target=None, name=None, args=(), kwargs={})

- target表示调用对象
- args表示调用对象的位置参数元组
- kwargs表示调用对象的字典
- name为别名
- group实质上不使用。

Process类常用方法：

- is_alive()： 判断进程实例是否还在执行；
- join([timeout])： 是否等待进程实例执行结束， 或等待多少秒；
- start()： 启动进程实例（ 创建子进程）；
- run()： 如果没有给定target参数， 对这个对象调用start()方法时， 就将执行对象中的run()方法；
- terminate()： 不管任务是否完成， 立即终止；

```python
# 代码2.2
#coding: utf-8
import os

from multiprocessing import Process
import os

# 进程要执行的代码
def run_proc(name):
    print('子线程 %s , ID是: %s' % (name, os.getpid()))


print('当前线程(父线程)的ID是:  %s' % os.getpid())
p = Process(target=run_proc, args=('test',))  # 创建Process的实例，并传入子线
p.start()  # 子线程开始执行
p.join()  # join方法用于线程间的同步，等线程执行完毕后再往下执行
print('子线程执行完毕，回到主线程%s' % os.getpid())
```

二、python 多进程

计算1+……+n的和

```python
#coding: utf-8
import time
def my_flow(n):
    total = 0
    for i in range(n):
        total += i
    print('1+...+%s=%s' % (n, total))
if __name__ == '__main__':
    start = time.time()
    myflow(5000000)
    myflow(6000000)
    end = time.time()
    print(end-start)
```

如上面的例子，我们进行两个计算任务，我们想加速, 多进程（进程池）实现。 python中实现多进程的方式有multiprocessing，concurrent.futures.ThreadPoolExecutor

1. multiprocessing

接下来了解下Process类

1.1 创建多进程

> - start 通过调用start方法启动进程，跟线程差不多
> - run：如果在创建Process对象的时候不指定target，那么就会默认执行Process的run方法
> - join 阻塞当前进程，直到调用join方法的那个进程执行完，再继续执行当前进程
> - mutilprocess.setDaemon(True) 守护进程就是不阻挡主程序退出，自己干自己的

```python
# 代码2.3
#coding: utf-8
from multiprocessing import Process
import time
def my_flow(n):
    total = 0
    for i in range(n):
        total += i
    print('1+...+%s=%s' % (n, total))

if __name__ == "__main__":
        start = time.time()
        print("main process run...")
        p1 = Process(target=my_flow, args=(5000000, ))       #target:指定进
        p2 = Process(target=my_flow, args=(6000000, ))

        p1.start()
        p2.start()
        p1.join()
        p2.join()
        #p1.join()
        #p2.join()
        print("main process runned all lines...")
        end = time.time()
        print('time cost {}'.format(end-start))
```

1.2 使用进程池

如果我们要启动大量的子进程且操作函数相同, 我们可以使用进程池的方式创建进程：

> - apply()主进程会被阻塞直到函数执行结束,apply_async()与apply用法一样，但它是非阻塞且支持结果返回进行回调。
> - close() 关闭pool，使其不在接受新的任务。
> - terminate() 结束工作进程，不在处理未完成的任务。
> - 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

```python
# 代码2.4
#coding: utf-8
import multiprocessing
import time

def my_flow(n):
    total = 0
    for i in range(n):
        total += i
    print('1+...+%s=%s' % (n, total))

if __name__ == "__main__":
    start = time.time()
    pool = multiprocessing.Pool(processes = 3)
    for i in [500000, 600000]:
        print( "第 %d 个 task" %(i))
        #维持执行的进程总数为processes，当一个进程执行完毕后会添加新的进程进去
        pool.apply(my_flow, (i, ))
        # 返回值
        # result = pool.apply(my_flow, (i, ))
        #
        # pool.apply_async(my_flow, (i, ))

    pool.close()
    pool.join()   #调用join之前，先调用close函数，否则会出错。执行完close后不会有新
    end = time.time()
    print('time cost {}'.format(end-start))
```

2.concurrent.futures.ProcessPoolExecutor

```python
# 代码2.5
import os
import time
from concurrent.futures import ProcessPoolExecutor as Pool
from concurrent.futures import as_completed

def my_flow(n):
    total = 0
    for i in range(n):
        total += i
    print('1+...+%s=%s' % (n, total))
    return (n, total)

if __name__ == "__main__":
    #
    start_time = time.time()
    result_list = []
    with Pool(max_workers=3) as executor:
        future_tasks = [executor.submit(my_flow, item) \
        for item in [500000, 600000, 700000]]

    result_list = as_completed(future_tasks)
    for r in result_list:
        if r.done():
            print(r.result())


    end_time = time.time()
    print("used time is ", end_time - start_time)
```

多进程、多线程框架

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。

先来看一段代码片段

```python
# 代码3.1
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

```python
# 代码3.2
# -*- 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
# 代码3.3
# -*- 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
# 代码3.4
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.