

---

# **python-parallel-programming- cookbook-cn Documentation**

**发布 1.0**

**laixintao**

**2017 年 11 月 22 日**



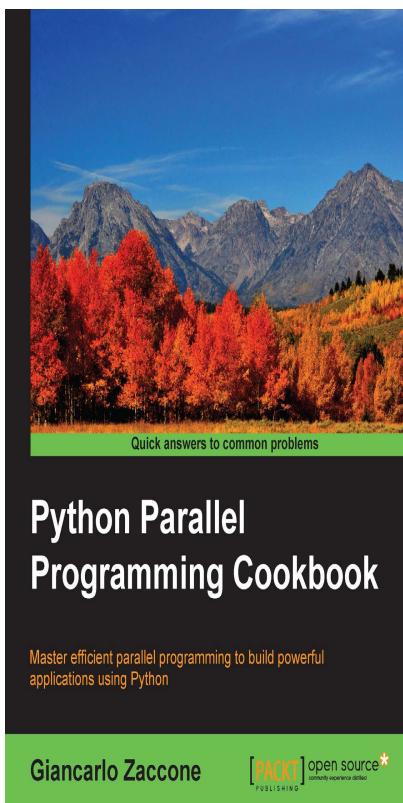
---

## Contents:

---

<b>1 第一章 认识并行计算和Python</b>	<b>3</b>
1.1 介绍 . . . . .	3
1.2 并行计算的内存架构 . . . . .	4
1.3 内存管理 . . . . .	4
1.4 并行编程模型 . . . . .	4
1.5 如何设计一个并行程序 . . . . .	4
1.6 如何评估并行程序的性能 . . . . .	4
1.7 介绍Python . . . . .	4
1.8 并行世界的Python . . . . .	7
1.9 介绍线程和进程 . . . . .	7
1.10 开始在Python中使用进程 . . . . .	7
1.11 开始在Python中使用线程 . . . . .	9
<b>2 第二章 基于线程的并行</b>	<b>13</b>
2.1 介绍 . . . . .	13
2.2 使用Python的线程模块 . . . . .	13
2.3 如何定义一个线程 . . . . .	14
2.4 如何确定当前的线程 . . . . .	15
2.5 如何实现一个线程 . . . . .	17
2.6 使用Lock进行线程同步 . . . . .	19
2.7 使用RLock进行线程同步 . . . . .	22
2.8 使用信号量进行线程同步 . . . . .	24
2.9 使用条件进行线程同步 . . . . .	27
2.10 使用事件进行线程同步 . . . . .	30
2.11 使用with语法 . . . . .	33
2.12 使用queue进行线程通信 . . . . .	36
2.13 评估多线程应用的性能 . . . . .	39
<b>3 第三章 基于进程的并行</b>	<b>45</b>
3.1 介绍 . . . . .	45
3.2 如何产生一个进程 . . . . .	46
3.3 如何为一个进程命名 . . . . .	48
3.4 如何在后台运行一个进程 . . . . .	49
3.5 如何杀掉一个进程 . . . . .	50
3.6 如何在子类中使用进程 . . . . .	51
3.7 如何在进程之间交换对象 . . . . .	52
3.8 进程如何同步 . . . . .	57

3.9	如何在进程之间管理状态 . . . . .	59
3.10	如何使用进程池 . . . . .	61
3.11	使用Python的mpi4py模块 . . . . .	62
3.12	点对点通讯 . . . . .	65
3.13	避免死锁问题 . . . . .	68
3.14	使用broadcast通讯 . . . . .	70
3.15	使用scatter通讯 . . . . .	70
3.16	使用gather通讯 . . . . .	70
3.17	使用Alltoall通讯 . . . . .	70
3.18	简化操作 . . . . .	70
3.19	如何优化通讯 . . . . .	70
<b>4</b>	<b>第四章 异步编程</b>	<b>71</b>
4.1	介绍 . . . . .	71
4.2	使用Python的 concurrent.futures 模块 . . . . .	72
4.3	使用Asyncio管理事件循环 . . . . .	75
4.4	使用Asyncio管理协程 . . . . .	79
4.5	使用Asyncio控制任务 . . . . .	83
4.6	使用Asyncio和Futures . . . . .	86
<b>5</b>	<b>第五章 分布式Python编程</b>	<b>89</b>
5.1	介绍 . . . . .	90
5.2	使用Celery实现分布式任务 . . . . .	90
5.3	如何使用Celery创建任务 . . . . .	90
5.4	使用SCOOP进行科学计算 . . . . .	90
5.5	使用SCOOP处理map函数 . . . . .	90
5.6	使用Pyro4进行远程方法调用 . . . . .	90
5.7	使用Pyro4清理对象 . . . . .	90
5.8	使用Pyro4部署客户端-服务器应用 . . . . .	90
5.9	使用PyCSP交流顺序的进程 . . . . .	90
5.10	使用Disco进行MapReduce . . . . .	90
5.11	使用RPyC远程调用 . . . . .	90
<b>6</b>	<b>第六章 Python GPU编程</b>	<b>93</b>
6.1	介绍 . . . . .	94
6.2	使用PyCUDA模块 . . . . .	94
6.3	如何创建一个PyCUDA应用 . . . . .	94
6.4	理解PyCuDA内存模型 . . . . .	94
6.5	使用GPUArray进行内核调用 . . . . .	94
6.6	使用PyCUDA评估元素 . . . . .	94
6.7	使用PyCUDA进行MapReduce操作 . . . . .	94
6.8	使用NumbaPro进行GPU编程 . . . . .	94
6.9	使用GPU加速的库 . . . . .	94
6.10	使用PyOpenCL模块 . . . . .	94
6.11	如何创建一个PyOpenCL应用 . . . . .	94
6.12	使用PyOpenCL评估元素 . . . . .	94
6.13	使用PyOpenCL测试你的GPU应用 . . . . .	94
<b>7</b>	<b>Indices and tables</b>	<b>95</b>





# CHAPTER 1

---

## 第一章 认识并行计算和Python

---

### 1.1 介绍

本章将介绍一些并行编程的架构和编程模型。对于初次接触并行编程技术的程序员来说，这些都是非常有用的概念；对于经验丰富的程序员来说，本章可以作为基础参考。本章中讲述了并行编程的两种解释，第一种解释是基于系统架构的，第二种解释基于程序示例F。并行编程对程序员来说一直是一项挑战。本章讨论并行程序的设计方法的时候，深入讲了这种编程方法。本章最后简单介绍了Python编程语言。Pyhton的易用和易学、可扩展性和丰富的库以及应用，让它成为了一个全能性的工具，当然，在并行计算方面也得心应手。最后结合在Python中的应用讲了线程和进程。解决一个大问题的一般方法是，将其拆分成若干小的、独立的问题，然后分别解它们。并行的程序也是使用这种方法，用多个处理器同时工作，来完成同一个任务。每一个处理器都做自己的那部分工作（独立的部分）。而且计算过程中处理器之间可能需要交换数据。如果，软件应用要求越来越高的计算能力。提高计算能力有两种思路：提高处理器的时钟速度或增加芯片上的核心数。提高时钟速度就必然会增加散热，然后每瓦特的性能就会降低，甚至可能要求特殊的冷却设备。提高芯片的核心数是更可行的一种方案，因为能源的消耗和散热，第一种方法必然有上限，而且计算能力提高没有特别明显。

为了解决这个问题，计算机硬件供应商的选择是多核心的架构，就是在同一个芯片上放两个或者多个处理器（核心）。GPU制造商也逐渐引进了这种基于多处理器核心的硬件架构。事实上，今天的计算机几乎都是各种多核、异构的计算单元组成的，每一个单元都有多个处理核心。

所以，对我们来说充分利用计算资源就显得至关重要，例如并行计算的程序、技术和工具等。

## 1.2 并行计算的内存架构

## 1.3 内存管理

## 1.4 并行编程模型

## 1.5 如何设计一个并行程序

## 1.6 如何评估并行程序的性能

## 1.7 介绍Python

Python是一种动态的解释型语言，应用场景广泛。它具有以下特性：

- 简明、易读的语法
- 丰富的标准库。通过第三方的软件模块，我们可以方便地添加数据类型、函数和对象
- 上手简单，开发和调试速度快。Python代码的快发速度可能比C/C++快10倍
- 基于Exception的错误处理机制
- 强大的自省功能
- 丰富的文档和活跃的社区

Python也可以作为一种胶水语言。通过Python，擅长不同编程语言的程序员可以在同一个项目中合作。例如开发一个数据型的应用时，C/C++程序员可以从底层实现高效的数值计算算法，而数据科学家可以通过Python调用这些算法，而不用花时间去学习底层的编程语言，C/C++程序员也不需要去理解科学数据层面的东西。

你可以从这里查看更多相关内容: <https://www.python.org/doc/essays/omg-darpa-mcc-position/>

### 1.7.1 准备工作

Python可以从这里下载: <https://www.python.org/downloads/>

虽然用NotePad或TextEdit就可以写Python代码，但是如果用集成开发环境（Integrated Development Environment, IDE）的话，编辑和调试会更方便。

目前已经有很多专门为Python设计的IDE，包括IDEL（<https://docs.python.org/3/library/idle.html>），PyCharm（<https://www.jetbrains.com/pycharm/>），Sublime Textd（<https://www.sublimetext.com/>）等。

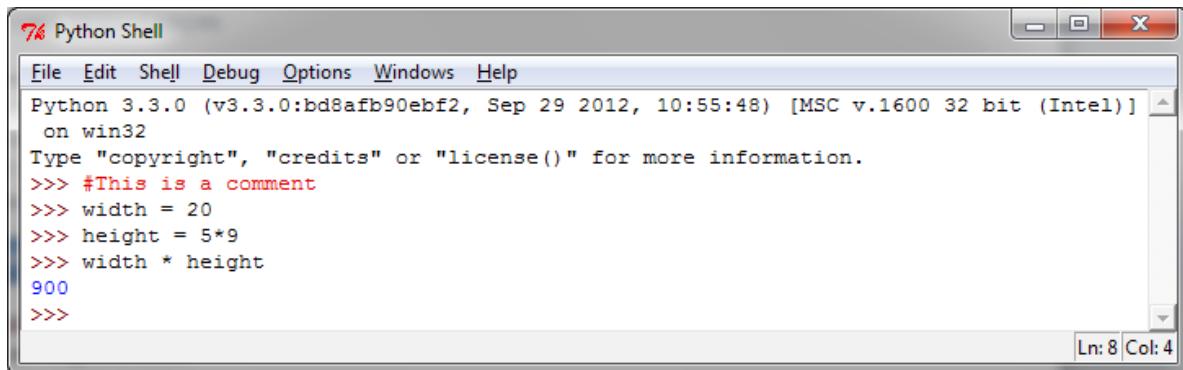
### 1.7.2 如何做...

下面来通过一些简短的代码熟悉一下Python。>>> 符号是Python解释器的提示符。

- 整数类型的操作:

```
>>> # This is a comment
>>> width = 20
>>> height = 5*9
>>> width * height
900
```

鉴于这是我们第一次展示代码，下面贴一下代码在Python解释器中的样子：



下面来看一下其他的例子：

- 复数：

```
>>> a=1.5+0.5j
>>> a.real
1.5
>>> a.imag
0.5
>>> abs(a)    # sqrt(a.real**2 + a.imag**2)
5.0
```

- 字符串操作：

```
>>> word = 'Help' + 'A' >>> word
'HelpA'
>>> word[4]
'A'
>>> word[0:2]
'He'
>>> word[-1]  # 最后一个字符
'A'
```

- 列表（list）操作：

```
>>> a = ['spam', 'eggs', 100, 1234] >>> a[0]
'spam'
>>> a[3]
1234
>>> a[-2]
100
>>> a[1:-1]
['eggs', 100]
>>> len(a)
4
```

- while 循环：

```
# Fibonacci series:
>>> while b < 10:
...     print b
...     a, b = b, a+b
...
1
1
2
3
5
8
```

- if 命令: 首先我们用 `input()` 从键盘读入一个整数:

```
>>>x = int(input("Please enter an integer here: "))
Please enter an integer here:
```

然后在输入的数字中使用 `if` 进行判断:

```
>>>if x < 0:
...     print ('the number is negative')
...elif x == 0:
...     print ('the number is zero')
...elif x == 1:
...     print ('the number is one')
...else:
...     print ('More')
...:
```

- for 循环: :

```
>>> # Measure some strings:
... a = ['cat', 'window', 'defenestrate'] >>> for x in a:
...     print (x, len(x))
...
cat 3
window 6
defenestrate 12
```

- 定义函数:

```
>>> def fib(n):      # 生成n以内的菲波那切数列
...         """Print a Fibonacci series up to n."""
...         a, b = 0, 1
...         while b < n:
...             print (b),
...             a, b = b, a+b
>>> # Now call the function we just defined:
...     fib(2000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

- 导入模块:

```
>>> import math
>>> math.sin(1)
0.8414709848078965
>>> from math import *
```

```
>>> log(1)
0.0
```

- 定义类:

```
>>> class Complex:
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
...         self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

## 1.8 并行世界的Python

作为一种解释型的语言，Python的速度并不算慢。如果对速度有很高的要求的话，可以选择用更块的语言实现，比如C或C++，然后用Python调用。Python的一种常见应用场景是实现高级的逻辑。Python的解释器就是用C语言写的，即CPython。解释器将Python转换成一种中间语言，叫做Python字节码，类似于汇编语言，但是包含一些更高级的指令。当一个运行一个Python程序的时候，评估循环不断将Python字节码转换成机器码。解释型语言的好处是方便编程和调试，但是程序的运行速度慢。其中的一种解决办法是，用C语言实现一些第三方的库，然后在Python中使用。另一种方法是使用即时编译器来替换Cpython，例如PyPy，PyPy对代码生成和Python的运行速度做了优化。但是在本书中，我们将研究第三种方法。Python提供了很多可以利用并行的模块，在后面的章节中，我们将着重讨论这些并行编程的模块。

接下来，本章将介绍两种基本概念：线程和进程，以及它们在Python中的表现。

## 1.9 介绍线程和进程

进程是应用程序的一个执行实例，比如，在桌面上双击浏览器图标将会运行一个浏览器。线程是一个控制流程，可以在进程内与其他活跃的线程同时执行。“控制流程”指的是顺序执行一些机器指令。进程可以包含多个线程，所以开启一个浏览器，操作系统将创建一个进程，并开始执行这个进程的主线程。每一个线程将独立执行一系列的指令（通常就是一个函数），并且和其他线程并行执行。然而，同一个进程内的线程可以共享一些地址空间和数据结构。线程也被称作“轻量进程”，因为它和进程有许多共同点，比如都是可以和其他控制流程同时运行的控制流程，说它“轻量”是因为实现一个进程比线程要繁重的多。重申一遍，不同于进程，多个线程可以共享很多资源，特别是地址空间和数据结构等。

总结一下：

- 进程可以包含多个并行运行的线程。
- 通常，操作系统创建和管理线程比进程更能节省CPU的资源。线程用于一些小任务，进程用于繁重的任务——运行应用程序。
- 同一个进程下的线程共享地址空间和其他资源，进程之间相互独立。

在深入研究通过线程和进程管理并行的Python模块之前，我们先来看一下Python中是如何使用这两者的。

## 1.10 开始在Python中使用进程

在大多数操作系统中，每个程序在一个进程中运行。通常，我们通过双击一个应用程序的图标来启动程

序。在本节中，我们简单地展示如何在Python中开启一个进程。进程有自己的地址空间，数据栈和其他的辅助数据来追踪执行过程；系统会管理所有进程的执行，通过调度程序来分配计算资源等。

### 1.10.1 准备工作

在第一个程序中，你需要先确保安装了Python。

最新的Python可以从 <https://www.python.org/> 下载安装。

### 1.10.2 如何做...

执行第一个示例，我们需要敲入下面两个代码文件：

- called\_Process.py
- calling\_Process.py

你可以使用Python IDE(3.3.0)来编辑下面的文件：

called\_Process 的代码如下：

```
print("Hello Python Parallel Cookbook!!")
closeInput = raw_input("Press ENTER to exit")
print "Closing calledProcess"
```

calling\_Process 的代码如下：

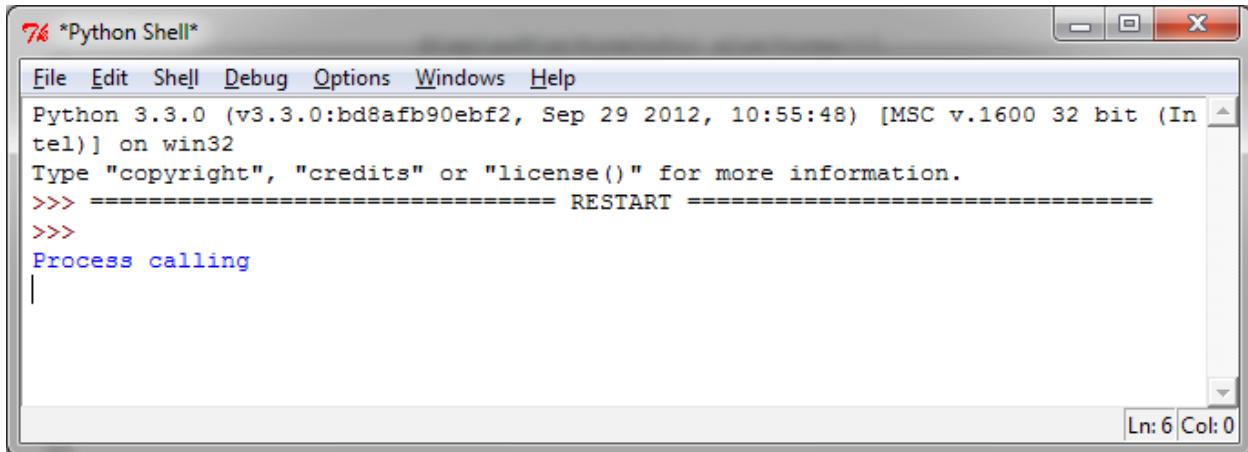
```
## The following modules must be imported
import os
import sys

## this is the code to execute
program = "python"
print("Process calling")
arguments = ["called_Process.py"]

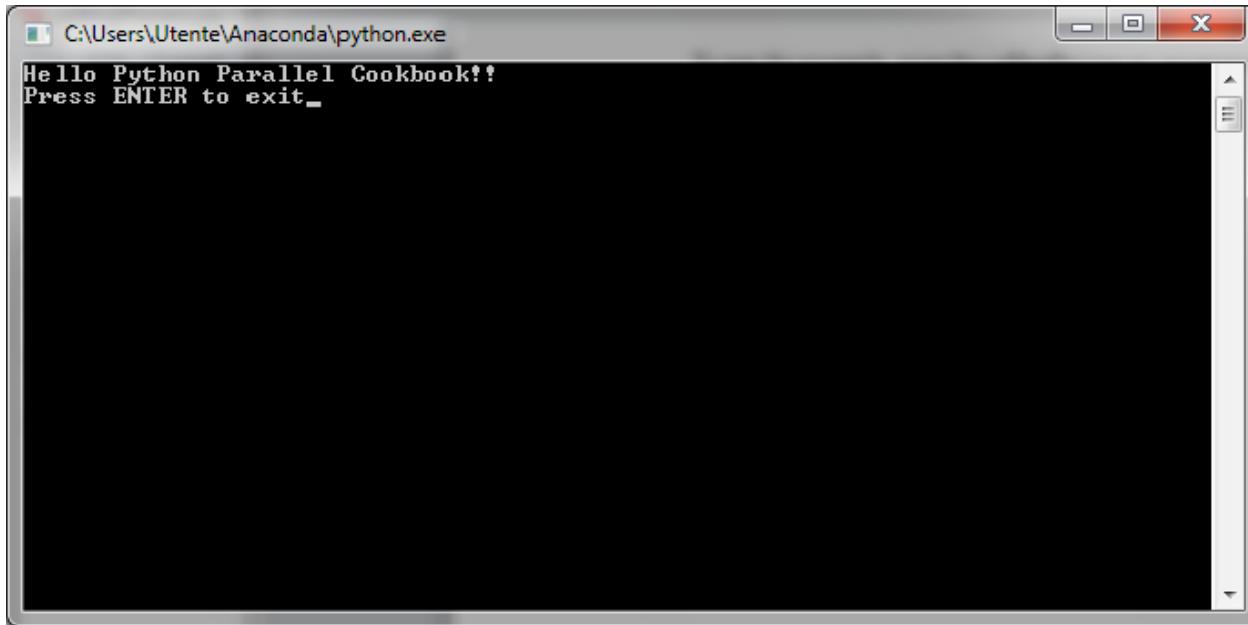
## we call the called_Process.py script
os.execvp(program, (program,) + tuple(arguments))
print("Good Bye!!")
```

运行例子的方法是，用Python IDE打开 calling\_Process 程序然后按下F5。

在Python shell看到的输出如下：



同时，系统的终端将看到如下输出：



我们有两个进程运行，按下Enter可以关闭系统终端。

### 1.10.3 如何做...

在前面的例子中，`execvp` 函数开启了一个新的进程，替换了当前的进程。注意“Good Bye”永远不会打印出来。相反，它会在当前的系统路径中搜索 `called_Process`，将第二个参数的内容作为独立的变量传给程序，然后在当前环境上下文中执行。`called_Process` ``中的 ``input()` 仅仅用来管理当前系统的闭包。本节展示了基于进程的并行，我们在后面会介绍更多通过进程（multiprocessing模块）管理并行的方法。

## 1.11 开始在Python中使用线程

如前面章节提到的那样，基于线程的并行是编写并行程序的标准方法。然而，Python解释器并不完全是线程安全的。为了支持多线程的Python程序，CPython使用了一个叫做全局解释器锁（Global Interpreter Lock， GIL）的技术。这意味着同一时间只有一个线程可以执行Python代码；执行某一个线程一小段时间之

后，Python会自动切换到下一个线程。GIL并没有完全解决线程安全的问题，如果多个线程试图使用共享数据，还是可能导致未确定的行为。

在本节中，我们将展示如何在Python程序中创建一个线程。

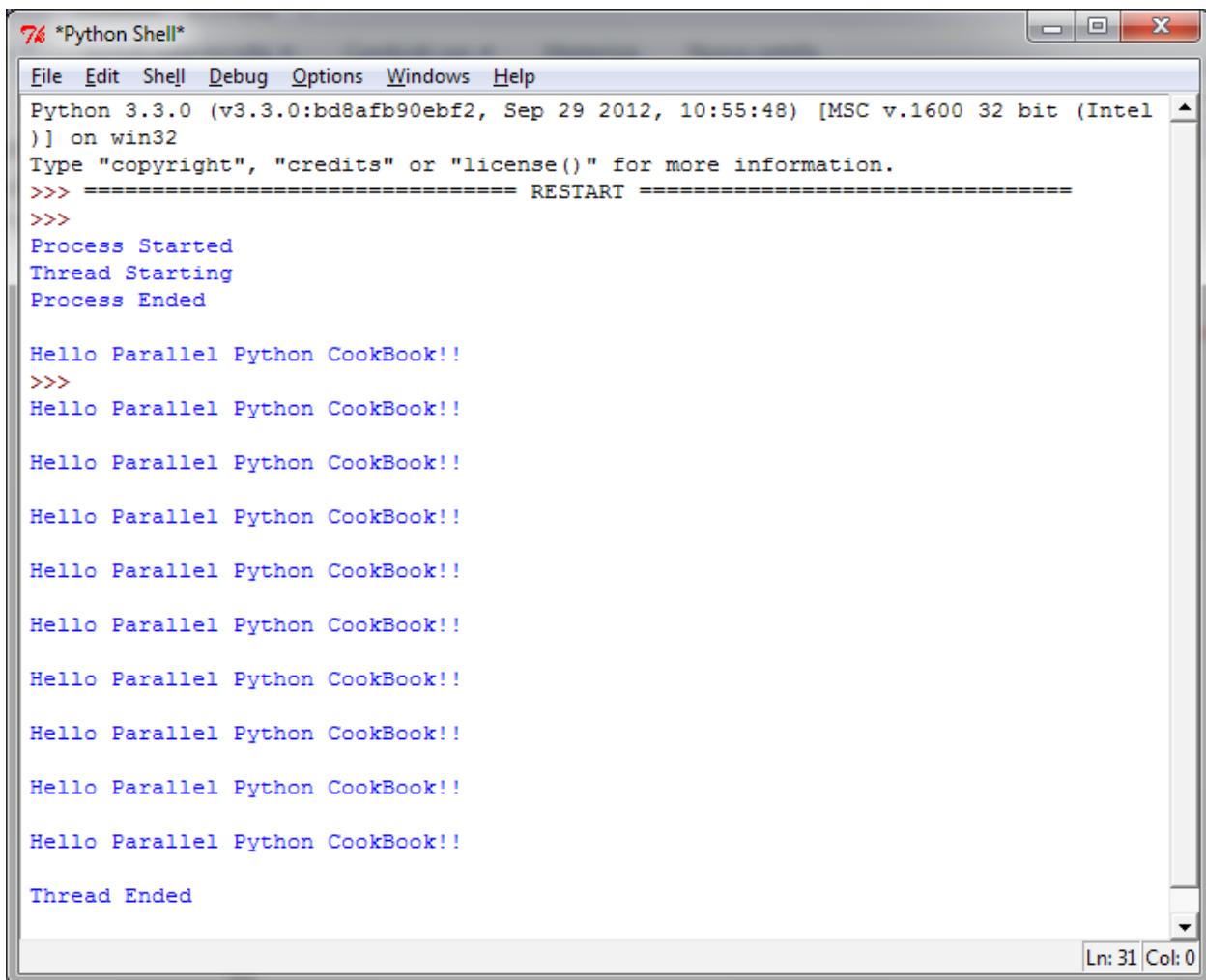
### 1.11.1 如何做...

我们需要 helloPythonWithThreads.py 来执行第一个例子：

```
# To use threads you need import Thread using the following code:  
from threading import Thread  
# Also we use the sleep function to make the thread "sleep"  
from time import sleep  
  
# To create a thread in Python you'll want to make your class work as a thread.  
# For this, you should subclass your class from the Thread class  
class CookBook(Thread):  
    def __init__(self):  
        Thread.__init__(self)  
        self.message = "Hello Parallel Python CookBook!\\n"  
  
    # this method prints only the message  
    def print_message(self):  
        print(self.message)  
  
    # The run method prints ten times the message  
    def run(self):  
        print("Thread Starting\\n")  
        x = 0  
        while (x < 10):  
            self.print_message()  
            sleep(2)  
            x += 1  
        print("Thread Ended\\n")  
  
# start the main process  
print("Process Started")  
  
# create an instance of the HelloWorld class  
hello_Python = CookBook()  
  
# print the message...starting the thread  
hello_Python.start()  
  
# end the main process  
print("Process Ended")
```

运行上面的代码，需要用Python IDE打开 calling\_Process.py 然后按下 F5.

在Python shell中你将看到以下输出：



The screenshot shows a Python Shell window with the title "74 \*Python Shell\*". The window contains the following text:

```
File Edit Shell Debug Options Windows Help
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:55:48) [MSC v.1600 32 bit (Intel
)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
Process Started
Thread Starting
Process Ended

Hello Parallel Python CookBook!!
>>>
Hello Parallel Python CookBook!!

Hello Parallel Python CookBook!!
Hello Parallel Python CookBook!!
Hello Parallel Python CookBook!!
Hello Parallel Python CookBook!!
Hello Parallel Python CookBook!!
Hello Parallel Python CookBook!!
Hello Parallel Python CookBook!!
Hello Parallel Python CookBook!!
Hello Parallel Python CookBook!!
Thread Ended

Ln: 31 Col: 0
```

## 1.11.2 讨论

主程序执行结束的时候，线程依然会每个两秒钟就打印一次信息。此例子证实了线程是在父进程中执行的一个子任务。

需要注意的一点是，永远不要留下任何线程在后台默默运行。否则在大型程序中这将给你带来无限痛苦。



## 第二章 基于线程的并行

---

### 2.1 介绍

目前，在软件应用中使用最广泛的并发编程范例是多线程。通常，一个应用有一个进程，分成多个独立的线程，并行运行、互相配合，执行不同类型的任务。

虽然这种模式存在一些缺点，有很多潜在的问题，但是多线程的应用依然非常广泛。

现在几乎所有的操作系统都支持多线程，几乎所有的编程语言都有相应的多线程机制，可以在应用中通过线程实现并发。

所以，使用多线程编程来实现并发的并用是个不错的选择。然而，多线程并不是唯一的选择，有不少其他的方案的表现比多线程好的多。

线程是独立的处理流程，可以和系统的其他线程并行或并发地执行。多线程可以共享数据和资源，利用所谓的共享内存空间。线程和进程的具体实现取决于你要运行的操作系统，但是总体来讲，我们可以说线程是包含在进程中的，同一进程的多个不同的线程可以共享相同的资源。相比而言，进程之间不会共享资源。

每一个线程基本上包含3个元素：程序计数器，寄存器和栈。与同一进程的其他线程共享的资源基本上包括数据和系统资源。每一个线程也有自己的运行状态，可以和其他线程同步，这点和进程一样。线程的状态大体上可以分为ready,running,blocked。线程的典型应用是应用软件的并行化——为了充分利用现代的多核处理器，使每个核心可以运行单个线程。相比于进程，使用线程的优势主要是性能。相比之下，在进程之间切换上下文要比在统一进程的多线程之间切换上下文要重的多。

多线程编程一般使用共享内容空间进行线程间的通讯。这就使管理内容空间成为多线程编程的重点和难点。

### 2.2 使用Python的线程模块

Python通过标准库的 `threading` 模块来管理线程。这个模块提供了很多不错的特性，让线程变得无比简单。实际上，线程模块提供了几种同时运行的机制，实现起来非常简单。

线程模块的主要组件如下：

- 线程对象
- Lock对象
- RLock对象
- 信号对象
- 条件对象
- 事件对象

在接下来的子章节中，我们将通过例子尝试这些由线程库提供的特性。以下实例基于Python 3.3（兼容Python 2.7）。

## 2.3 如何定义一个线程

使用线程最简单的一个方法是，用一个目标函数实例化一个Thread然后调用 `start()` 方法启动它。Python的threading模块提供了 Thread() 方法在不同的线程中运行函数或处理过程等。

```
class threading.Thread(group=None,
                      target=None,
                      name=None,
                      args=(),
                      kwargs={})
```

上面的代码中：

- group: 一般设置为 `None`，这是为以后的一些特性预留的
- target: 当线程启动的时候要执行的函数
- name: 线程的名字，默认会分配一个唯一名字 `Thread-N`
- args: 传递给 target 的参数，要试用tuple类型
- kwargs: 同上，试用字段类型dict

创建线程的方法非常实用，通过参数和目标告诉线程应该做什么。下面这个例子传递一个数字给线程（这个数字正好等于线程号码），目标函数会打印出这个数字。

### 2.3.1 如何做...

让我们看一下如何通过threading模块创建线程，只需要几行代码就可以了：

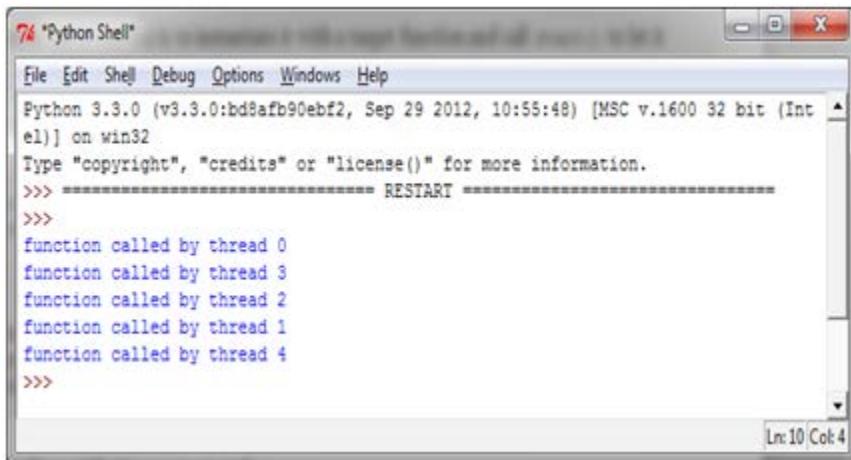
```
import threading

def function(i):
    print ("function called by thread %i\n" % i)
    return

threads = []

for i in range(5):
    t = threading.Thread(target=function , args=(i, ))
    threads.append(t)
    t.start()
    t.join()
```

上面的代码运行结果如下：



The screenshot shows a Windows-style application window titled "74 \*Python Shell". The menu bar includes File, Edit, Shell, Debug, Options, Windows, Help. The main window displays a Python 3.3.0 shell session. It starts with the Python version information: "Python 3.3.0 (v3.3.0:bd8af90ebf2, Sep 29 2012, 10:55:48) [MSC v.1600 32 bit (Intel)] on win32". It then shows the command ">>> ===== RESTART =====" followed by five lines of text: "function called by thread 0", "function called by thread 3", "function called by thread 2", "function called by thread 1", and "function called by thread 4". The bottom right corner of the window shows "Ln: 10 Col: 4".

注意，输出的顺序可能和上图不同。事实上，多个线程可能同时向 `stdout` 打印结果，所以输出顺序无法事先确定。

(译者注：这段代码怀疑存在错误，因为写了 `t.join()`，这意味着，`t`线程结束之前并不会看到后续的线程，换句话说，主线程会调用`t`线程，然后等待`t`线程完成再执行for循环开启下一个`t`线程，事实上，这段代码是顺序运行的，实际运行顺序永远是01234顺序出现，不会出现图中结果)

## 2.3.2 讨论

导入内置`threading`模块，简单地使用`pyhton`命令就可以了：

```
import threading
```

在主程序中，我们使用目标函数 `function` 初始化了一个线程对象 `Thread`。同时还传入了用于打印的一个参数：

```
t = threading.Thread(target=function , args=(i, ))
```

线程被创建之后并不会马上运行，需要手动调用 `start()`，`join()` 让调用它的线程一直等待直到执行结束（即阻塞调用它的主线程，`t` 线程执行结束，主线程才会继续执行）：

```
t.start()
t.join()
```

## 2.4 如何确定当前的线程

使用参数来确认或命名线程是笨拙且没有必要的。每一个 `Thread` 实例创建的时候都有一个带默认值的名字，并且可以修改。在服务端通常一个服务进程都有多个线程服务，负责不同的操作，这时候命名线程是很实用的。

### 2.4.1 如何做...

为了演示如何确定正在运行的线程，我们创建了三个目标函数，并且引入了 `time` 在运行期间挂起2s，让结果更明显。

```

import threading
import time

def first_function():
    print(threading.currentThread().getName() + str(' is Starting '))
    time.sleep(2)
    print (threading.currentThread().getName() + str(' is Exiting '))
    return

def second_function():
    print(threading.currentThread().getName() + str(' is Starting '))
    time.sleep(2)
    print (threading.currentThread().getName() + str(' is Exiting '))
    return

def third_function():
    print(threading.currentThread().getName() + str(' is Starting '))
    time.sleep(2)
    print (threading.currentThread().getName() + str(' is Exiting '))
    return

if __name__ == "__main__":
    t1 = threading.Thread(name='first_function', target=first_function)
    t2 = threading.Thread(name='second_function', target=second_function)
    t3 = threading.Thread(name='third_function', target=third_function)
    t1.start()
    t2.start()
    t3.start()

```

输出如下图所示:

The screenshot shows the Python Shell window with the title "76 Python Shell". The window displays the following output:

```

Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:55:48) [MSC v.1600 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
first_function is Starting
second_function is Starting
third_function is Starting

first_function is Exiting
second_function is Exiting
third_function is Exiting

>>> |

```

The window has a status bar at the bottom right indicating "Ln: 17 Col: 4".

## 2.4.2 讨论

我们使用目标函数实例化线程。同时，我们传入 name 参数作为线程的名字，如果不传这个参数，将使用默认的参数：

```
t1 = threading.Thread(name='first_function', target=first_function)
t2 = threading.Thread(name='second_function', target=second_function)
t3 = threading.Thread(target=third_function)
```

(译者注：这里的代码和上面的不一样，可能作者本意是第三个线程不加参数来测试默认的行为，如果改为这里的代码，那么线程3将会输出的是 Thread-1 is Starting 以及 Thread-1 is Exiting，读者可以自行尝试)

最后调用 `start()` 和 `join()` 启动它们。

```
t1.start()
t2.start()
t3.start()
t1.join()
t2.join()
t3.join()
```

(译者注：同样，这里也是一个错误。上面的源代码并没有调用 `join()`，那么将以不确定的顺序结束。如果按照这里的代码，那么t1,t2,t3必将以固定的顺序结束)

## 2.5 如何实现一个线程

使用`threading`模块实现一个新的线程，需要下面3步：

- 定义一个 `Thread` 类的子类
- 覆盖 `__init__(self [,args])` 方法，可以添加额外的参数
- 最后，需要覆盖 `run(self, [,args])` 方法来实现线程要做的事情

当你创建了新的 `Thread` 子类的时候，你可以实例化这个类，调用 `start()` 方法来启动它。线程启动之后将会执行 `run()` 方法。

### 2.5.1 如何做...

为了在子类中实现线程，我们定义了 `myThread` 类。其中有两个方法需要手动实现：

```
import threading
import time

exitFlag = 0

class myThread (threading.Thread):
    def __init__(self, threadID, name, counter):
        threading.Thread.__init__(self)
        self.threadID = threadID
        self.name = name
        self.counter = counter

    def run(self):
        print("Starting " + self.name)
        print_time(self.name, self.counter, 5)
        print("Exiting " + self.name)

    def print_time(threadName, delay, counter):
        while counter:
```

```

if exitFlag:
    # 译者注: 原书中使用的thread, 但是Python3中已经不能使用thread, 以_thread取代, 因此应该
    # import _thread
    # _thread.exit()
    thread.exit()
    time.sleep(delay)
    print("%s: %s" % (threadName, time.ctime(time())))
    counter -= 1

# Create new threads
thread1 = myThread(1, "Thread-1", 1)
thread2 = myThread(2, "Thread-2", 2)

# Start new Threads
thread1.start()
thread2.start()

# 以下两行为译者添加, 如果要获得和图片相同的结果,
# 下面两行是必须的。疑似原作者的疏漏
thread1.join()
thread2.join()
print("Exiting Main Thread")

```

上面的代码输出结果如下:

```

76 Python Shell
File Edit Shell Debug Options Windows Help
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:55:48) [MSC v.1600 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
Starting Thread-1
Starting Thread-2

Thread-1: Sun Apr 12 15:42:00 2015
Thread-2: Sun Apr 12 15:42:01 2015
Thread-1: Sun Apr 12 15:42:01 2015
Thread-1: Sun Apr 12 15:42:02 2015
Thread-2: Sun Apr 12 15:42:03 2015
Thread-1: Sun Apr 12 15:42:03 2015
Thread-1: Sun Apr 12 15:42:04 2015
Exiting Thread-1

Thread-2: Sun Apr 12 15:42:05 2015
Thread-2: Sun Apr 12 15:42:07 2015
Thread-2: Sun Apr 12 15:42:09 2015
Exiting Thread-2

Exiting Main Thread
>>>

```

## 2.5.2 讨论

线程模块是创建和管理线程的首选形式。每一个线程都通过一个集成 Thread 的子类代表，覆盖 run() 方法来实现逻辑，这个方法是线程的入口。在主程序中，我们创建了多个 myThread 的类型实例，然后执行

`start()` 方法启动它们。调用 `Thread` 的构造器是必须的，通过它我们可以给线程定义一些名字或分组之类的属性。调用 `start()` 之后线程变为活跃状态，并且持续直到 `run()` 结束，或者中间出现异常。所有的线程都执行完成之后，程序结束。

`join()` 命令控制主线程的终止。

## 2.6 使用Lock进行线程同步

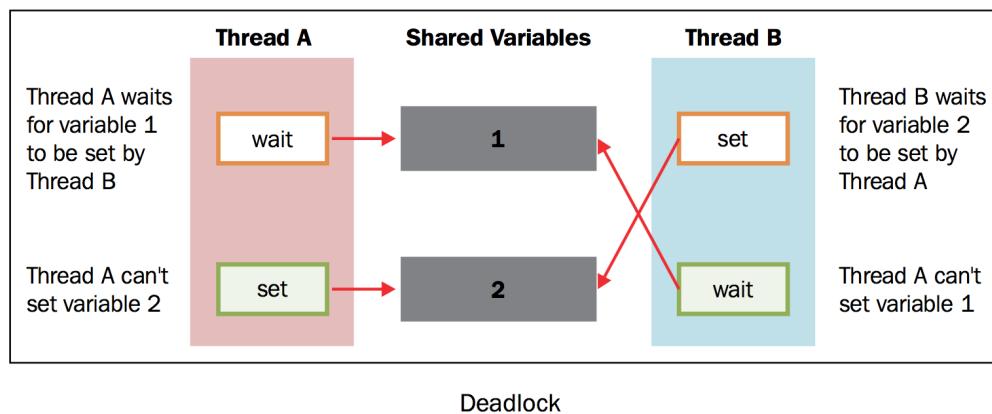
当两个或以上对共享内存的操作发生在并发线程中，并且至少有一个可以改变数据，又没有同步机制的条件下，就会产生竞争条件，可能会导致执行无效代码、bug、或异常行为。

竞争条件最简单的解决方法是使用锁。锁的操作非常简单，当一个线程需要访问部分共享内存时，它必须先获得锁才能访问。此线程对这部分共享资源使用完成之后，该线程必须释放锁，然后其他线程就可以拿到这个锁并访问这部分资源了。

很显然，避免竞争条件出现是非常重要的，所以我们要保证，在同一时刻只有一个线程允许访问共享内存。

尽管原理很简单，但是这样使用是work的。

然而，在实际使用的过程中，我们发现这个方法经常会导致一种糟糕的死锁现象。当不同的线程要求得到一个锁时，死锁就会发生，这时程序不可能继续执行，因为它们互相拿着对方需要的锁。



为了简化问题，我们设有两个并发的线程（**线程A** 和 **线程B**），需要 **资源1** 和 **资源2**。假设 **线程A** 需要 **资源1**，**线程B** 需要 **资源2**。在这种情况下，两个线程都使用各自的锁，目前为止没有冲突。现在假设，在双方释放锁之前，**线程A** 需要 **资源2** 的锁，**线程B** 需要 **资源1** 的锁，没有资源线程不会继续执行。鉴于目前两个资源的锁都是被占用的，而且在对方的锁释放之前都处于等待且不释放锁的状态。这是死锁的典型情况。所以如上所说，使用锁来解决同步问题是一个可行却存在潜在问题的方案。

本节中，我们描述了Python的线程同步机制，`lock()`。通过它我们可以将共享资源某一时刻的访问限制在单一线程或单一类型的线程上，线程必须得到锁才能使用资源，并且之后必须允许其他线程使用相同的资源。

### 2.6.1 如何做...

下面的例子展示了如何通过 `lock()` 管理线程。在下面的代码中，我们有两个函数：`increment()` 和 `decrement()`。第一个函数对共享资源执行加1的操作，另一个函数执行减1。两个函数分别使用线程封

装。除此之外，每一个函数都有一个循环重复执行操作。我们想要保证，通过对共享资源的管理，执行结果是共享资源最后等于初始值0。

代码如下：

```
# -*- coding: utf-8 -*-

import threading

shared_resource_with_lock = 0
shared_resource_with_no_lock = 0
COUNT = 100000
shared_resource_lock = threading.Lock()

# 有锁的情况
def increment_with_lock():
    global shared_resource_with_lock
    for i in range(COUNT):
        shared_resource_lock.acquire()
        shared_resource_with_lock += 1
        shared_resource_lock.release()

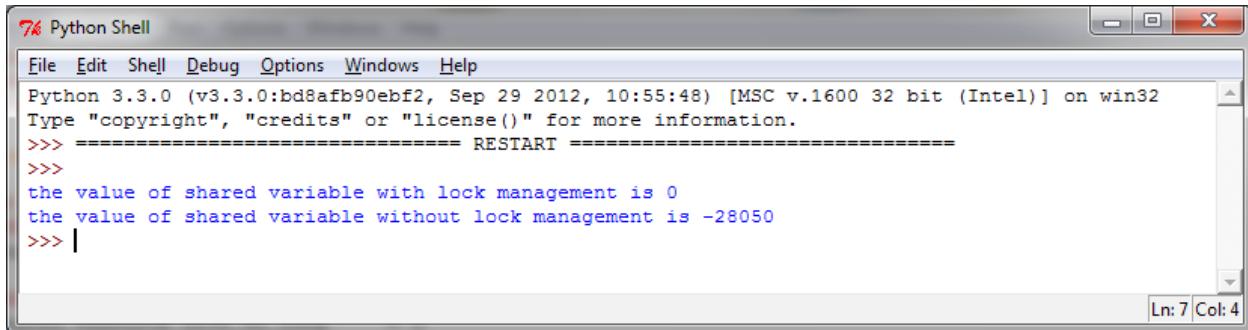
def decrement_with_lock():
    global shared_resource_with_lock
    for i in range(COUNT):
        shared_resource_lock.acquire()
        shared_resource_with_lock -= 1
        shared_resource_lock.release()

# 没有锁的情况
def increment_without_lock():
    global shared_resource_with_no_lock
    for i in range(COUNT):
        shared_resource_with_no_lock += 1

def decrement_without_lock():
    global shared_resource_with_no_lock
    for i in range(COUNT):
        shared_resource_with_no_lock -= 1

if __name__ == "__main__":
    t1 = threading.Thread(target=increment_with_lock)
    t2 = threading.Thread(target=decrement_with_lock)
    t3 = threading.Thread(target=increment_without_lock)
    t4 = threading.Thread(target=decrement_without_lock)
    t1.start()
    t2.start()
    t3.start()
    t4.start()
    t1.join()
    t2.join()
    t3.join()
    t4.join()
    print ("the value of shared variable with lock management is %s" % shared_
resource_with_lock)
    print ("the value of shared variable with race condition is %s" % shared_resource_
with_no_lock)
```

代码执行的结果如下：



可以看出，如果有锁来管理线程的话，我们会得到正确的结果。这里要注意，没有锁的情况下并不一定会得到错误的结果，但是重复执行多次，总会出现错误的结果。而有锁的情况下结果总会是正确的。

## 2.6.2 讨论

在主程序中，我们有以下步骤：

```
t1 = threading.Thread(target=increment_with_lock)
t2 = threading.Thread(target=decrement_with_lock)
```

启动线程：

```
t1.start()
t2.start()
```

然后阻塞主线程直到所有线程完成：

```
t1.join()
t2.join()
```

在 `increment_with_lock()` 函数和 `decrement_with_lock()` 函数中，可以看到我们使用了 `lock` 语句。当你需要使用资源的时候，调用 `acquire()` 拿到锁（如果锁暂时不可用，会一直等待直到拿到），最后调用 `release()`：

```
shared_resource_lock.acquire()
shared_resource_with_lock -= 1
shared_resource_lock.release()
```

让我们总结一下：

- 锁有两种状态： `locked`（被某一线程拿到）和 `unlocked`（可用状态）
- 我们有两个方法来操作锁： `acquire()` 和 `release()`

需要遵循以下规则：

- 如果状态是 `unlocked`，可以调用 `acquire()` 将状态改为 `locked`
- 如果状态是 `locked`，`acquire()` 会被 `block` 直到另一线程调用 `release()` 释放锁
- 如果状态是 `unlocked`，调用 `release()` 将导致 `RuntimError` 异常
- 如果状态是 `locked`，可以调用 `release()` 将状态改为 `unlocked`

## 2.6.3 了解更多

尽管理论上行得通，但是锁的策略不仅会导致有害的僵持局面。还会对应用程序的其他方面产生负面影响。这是一种保守的方法，经常会引起不必要的开销，也会限制程序的可扩展性和可读性。更重要的是，有时候需要对多进程共享的内存分配优先级，使用锁可能和这种优先级冲突。最后，从实践的经验来看，使用锁的应用将对debug带来不小的麻烦。所以，最好使用其他可选的方法确保同步读取共享内存，避免竞争条件。

## 2.7 使用RLock进行线程同步

如果你想让只有拿到锁的线程才能释放该锁，那么应该使用 `RLock()` 对象。和 `Lock()` 对象一样，`RLock()` 对象有两个方法：`acquire()` 和 `release()`。当你需要在类外面保证线程安全，又要在类内使用同样方法的时候 `RLock()` 就很实用了。

(译者注：RLock原作解释的太模糊了，译者在此擅自添加一段。RLock其实叫做“Reentrant Lock”，就是可以重复进入的锁，也叫做“递归锁”。这种锁对比Lock有是三个特点：1. 谁拿到谁释放。如果线程A拿到锁，线程B无法释放这个锁，只有A可以释放；2. 同一线程可以多次拿到该锁，即可以acquire多次；3. acquire多少次就必须release多少次，只有最后一次release才能改变RLock的状态为unlocked)

### 2.7.1 如何做...

在示例代码中，我们引入了 `Box` 类，有 `add()` 方法和 `remove()` 方法，提供了进入 `execute()` 方法的入口。`execute()` 的执行由 `RLock()` 控制：

```
import threading
import time

class Box(object):
    lock = threading.RLock()

    def __init__(self):
        self.total_items = 0

    def execute(self, n):
        Box.lock.acquire()
        self.total_items += n
        Box.lock.release()

    def add(self):
        Box.lock.acquire()
        self.execute(1)
        Box.lock.release()

    def remove(self):
        Box.lock.acquire()
        self.execute(-1)
        Box.lock.release()

## These two functions run n in separate
## threads and call the Box's methods
def adder(box, items):
    while items > 0:
        print("adding 1 item in the box")
        box.add()
        items -= 1
```

```

        time.sleep(1)
        items -= 1

def remover(box, items):
    while items > 0:
        print("removing 1 item in the box")
        box.remove()
        time.sleep(1)
        items -= 1

## the main program build some
## threads and make sure it works
if __name__ == "__main__":
    items = 5
    print("putting %s items in the box " % items)
    box = Box()
    t1 = threading.Thread(target=add, args=(box, items))
    t2 = threading.Thread(target=remover, args=(box, items))
    t1.start()
    t2.start()

    t1.join()
    t2.join()
    print("%s items still remain in the box " % box.total_items)

```

运行结果如下：

```

Python Shell
File Edit Shell Debug Options Windows Help
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:55:48) [MSC v.1600 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
putting 5 items in the box
adding 1 item in the box

removing 1 item in the box
adding 1 item in the box

removing 1 item in the box
adding 1 item in the box

removing 1 item in the box
adding 1 item in the box
removing 1 item in the box

adding 1 item in the box
removing 1 item in the box

0 items still remain in the box
>>>

```

## 2.7.2 讨论

主程序的代码几乎和之前的例子一样。两个线程 `t1` 和 `t2` 分别分配了 `add()` 函数和 `remover()` 函数。

当item的数量大于0的时候，函数工作。调用 RLock() 的位置是在 Box 类内：

```
class Box(object):
    lock = threading.RLock()
```

adder() 和 remover() 两个函数在 Box 类内操作items，即调用 Box 类的方法： add() 和 remove()。每一次方法调用，都会有一次拿到资源然后释放资源的过程。至于 lock() 对象， RLock() 对象有 acquire() 和 release() 方法可以拿到或释放资源；然后每一次方法调用中，我们都有以下操作：

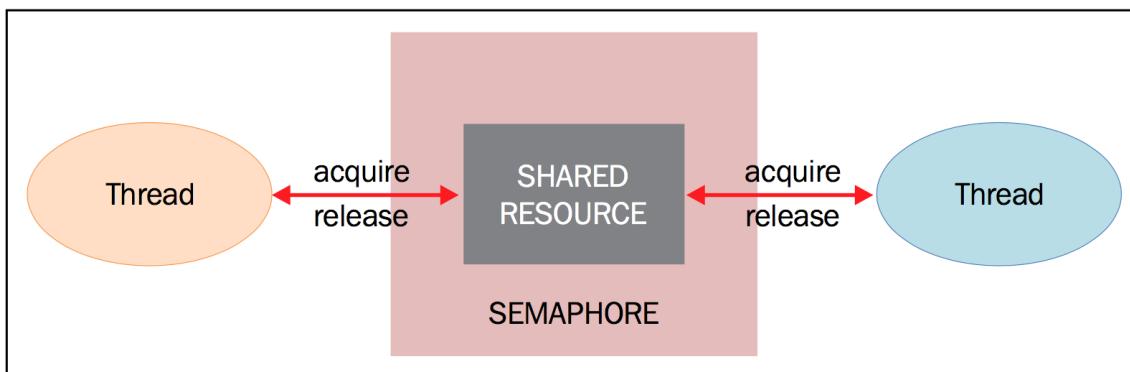
```
Box.lock.acquire()
# ...do something
Box.lock.release()
```

## 2.8 使用信号量进行线程同步

信号量由E.Dijkstra发明并第一次应用在操作系统中，信号量是由操作系统管理的一种抽象数据类型，用于在多线程中同步对共享资源的使用。本质上说，信号量是一个内部数据，用于标明当前的共享资源可以有多少并发读取。

同样的，在threading模块中，信号量的操作有两个函数，即 acquire() 和 release()，解释如下：

- 每当线程想要读取关联了信号量的共享资源时，必须调用 acquire()，此操作减少信号量的内部变量，如果此变量的值非负，那么分配该资源的权限。如果是负值，那么线程被挂起，直到有其他的线程释放资源。
- 当线程不再需要该共享资源，必须通过 release() 释放。这样，信号量的内部变量增加，在信号量等待队列中排在最前面的线程会拿到共享资源的权限。



Thread synchronization with semaphores

虽然表面上看信号量机制没什么明显的问题，如果信号量的等待和通知操作都是原子的，确实没什么问题。但如果不是，或者两个操作有一个终止了，就会导致糟糕的情况。

举个例子，假设有两个并发的线程，都在等待一个信号量，目前信号量的内部值为1。假设第线程A将信号量的值从1减到0，这时候控制权切换到了线程B，线程B将信号量的值从0减到-1，并且在这里被挂起等待，这时控制权回到线程A，信号量已经成为了负值，于是第一个线程也在等待。

这样的话，尽管当时的信号量是可以让线程访问资源的，但是因为非原子操作导致了所有的线程都在等待状态。

## 2.8.1 准备工作

下面的代码展示了信号量的使用，我们有两个线程，`producer()` 和 `consumer()`，它们使用共同的资源，即`item`。`producer()` 的任务是生产`item`，`consumer()` 的任务是消费`item`。

当`item`还没有被生产出来，`consumer()`一直等待，当`item`生产出来，`producer()` 线程通知消费者资源可以使用了。

## 2.8.2 如何做...

在以下的代码中，我们使用生产者-消费者模型展示通过信号量的同步。当生产者生产出`item`，便释放信号量。然后消费者拿到资源进行消费。

```
# -*- coding: utf-8 -*-

"""Using a Semaphore to synchronize threads"""
import threading
import time
import random

# The optional argument gives the initial value for the internal
# counter;
# it defaults to 1.
# If the value given is less than 0, ValueError is raised.
semaphore = threading.Semaphore(0)

def consumer():
    print("consumer is waiting.")
    # Acquire a semaphore
    semaphore.acquire()
    # The consumer have access to the shared resource
    print("Consumer notify : consumed item number %s" % item)

def producer():
    global item
    time.sleep(10)
    # create a random item
    item = random.randint(0, 1000)
    print("producer notify : produced item number %s" % item)
    # Release a semaphore, incrementing the internal counter by one.
    # When it is zero on entry and another thread is waiting for it
    # to become larger than zero again, wake up that thread.
    semaphore.release()

if __name__ == '__main__':
    for i in range (0,5) :
        t1 = threading.Thread(target=producer)
        t2 = threading.Thread(target=consumer)
        t1.start()
        t2.start()
        t1.join()
        t2.join()
    print("program terminated")
```

程序会运行5轮，结果如下：

The screenshot shows a Python Shell window with the title '76 Python Shell'. The window contains the following text:

```

Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:55:48) [MSC v.1600 32 bit (Intel)]
on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
consumer is waiting.
producer notify : produced item number 193
Consumer notify : consumed item number 193
consumer is waiting.
producer notify : produced item number 631
Consumer notify : consumed item number 631
consumer is waiting.
producer notify : produced item number 770
Consumer notify : consumed item number 770
consumer is waiting.
producer notify : produced item number 688
Consumer notify : consumed item number 688
consumer is waiting.
producer notify : produced item number 16
Consumer notify : consumed item number 16
program terminated
>>> |

```

The window has a status bar at the bottom right showing 'Ln: 21 Col: 4'.

### 2.8.3 讨论

信号量被初始化为0，此信号量唯一的目的是同步两个或多个线程。在这里，我们的线程必须并行运行，所以需要信号量同步：

```
semaphore = threading.Semaphore(0)
```

这个操作和lock中的机制非常相似，producer() 完成创建item之后，释放资源：

```
semaphore.release()
```

信号量的 release() 可以提高计数器然后通知其他的线程。同样的，consumer() 方法可以通过下面的方法拿到资源：

```
semaphore.acquire()
```

如果信号量的计数器到了0，就会阻塞 acquire() 方法，直到得到另一个线程的通知。如果信号量的计数器大于0，就会对这个值-1然后分配资源。

最后，拿到数据并打印输出：

```
print("Consumer notify : consumed item number %s" % item)
```

### 2.8.4 了解更多

信号量的一个特殊用法是互斥量。互斥量是初始值为1的信号量，可以实现数据、资源的互斥访问。

信号量在支持多线程的编程语言中依然应用很广，然而这可能导致死锁的情况。例如，现在有一个线程t1先等待信号量s1，然后等待信号量s2，而线程t2会先等待信号量s2，然后再等待信号量s1，这样就可能会发生死锁，导致t1等待s2，但是t2在等待s1。

## 2.9 使用条件进行线程同步

条件指的是应用程序状态的改变。这是另一种同步机制，其中某些线程在等待某一条件发生，其他的线程会在该条件发生的时候进行通知。一旦条件发生，线程会拿到共享资源的唯一权限。

### 2.9.1 准备工作

解释条件机制最好的例子还是生产者-消费者问题。在本例中，只要缓存不满，生产者一直向缓存生产；只要缓存不空，消费者一直从缓存取出（之后销毁）。当缓冲队列不为空的时候，生产者将通知消费者；当缓冲队列不满的时候，消费者将通知生产者。

### 2.9.2 如何做...

为了演示条件机制，我们将再一次使用生产者-消费者的例子：

```
from threading import Thread, Condition
import time

items = []
condition = Condition()

class consumer(Thread):
    def __init__(self):
        Thread.__init__(self)

    def consume(self):
        global condition
        global items
        condition.acquire()
        if len(items) == 0:
            condition.wait()
            print("Consumer notify : no item to consume")
        items.pop()
        print("Consumer notify : consumed 1 item")
        print("Consumer notify : items to consume are " + str(len(items)))

        condition.notify()
        condition.release()

    def run(self):
        for i in range(0, 20):
            time.sleep(2)
            self.consume()

class producer(Thread):
    def __init__(self):
        Thread.__init__(self)
```

```
def produce(self):
    global condition
    global items
    condition.acquire()
    if len(items) == 10:
        condition.wait()
        print("Producer notify : items produced are " + str(len(items)))
        print("Producer notify : stop the production!!")
    items.append(1)
    print("Producer notify : total items produced " + str(len(items)))
    condition.notify()
    condition.release()

def run(self):
    for i in range(0, 20):
        time.sleep(1)
        self.produce()

if __name__ == "__main__":
    producer = producer()
    consumer = consumer()
    producer.start()
    consumer.start()
    producer.join()
    consumer.join()
```

运行的结果如下：

```

Python Shell
File Edit Shell Debug Options Windows Help
Producer notify : total items produced 7
Consumer notify : consumed 1 item
Consumer notify : items to consume are 6
Producer notify : total items produced 7
Producer notify : total items produced 8
Consumer notify : consumed 1 item
Consumer notify : items to consume are 7
Producer notify : total items produced 8
Producer notify : total items produced 9
Consumer notify : consumed 1 item
Consumer notify : items to consume are 8
Producer notify : total items produced 9
Producer notify : total items produced 10
Consumer notify : consumed 1 item
Consumer notify : items to consume are 9
Producer notify : total items produced 10
Consumer notify : consumed 1 item
Consumer notify : items to consume are 9
Consumer notify : consumed 1 item
Consumer notify : items to consume are 8
Consumer notify : consumed 1 item
Consumer notify : items to consume are 7
Consumer notify : consumed 1 item
Consumer notify : items to consume are 6
Consumer notify : consumed 1 item
Consumer notify : items to consume are 5
Consumer notify : consumed 1 item
Consumer notify : items to consume are 4
Consumer notify : consumed 1 item
Consumer notify : items to consume are 3
Consumer notify : consumed 1 item
Consumer notify : items to consume are 2
Consumer notify : consumed 1 item
Consumer notify : items to consume are 1
Consumer notify : consumed 1 item
Consumer notify : items to consume are 0
>>>
Ln: 84 Col: 4

```

### 2.9.3 讨论

消费者通过拿到锁来修改共享的资源 `items[]` :

```
condition.acquire()
```

如果list的长度为0，那么消费者就进入等待状态:

```
if len(items) == 0:
    condition.wait()
```

否则就通过 `pop` 操作消费一个item:

```
items.pop()
```

然后，消费者的状态被通知给生产者，同时共享资源释放:

```
condition.notify()
condition.release()
```

生产者拿到共享资源，然后确认缓冲队列是否已满（在我们的这个例子中，最大可以存放10个item），如果已经满了，那么生产者进入等待状态，知道被唤醒:

```
condition.acquire()
if len(items) == 10:
    condition.wait()
```

如果队列没有满，就生产1个item，通知状态并释放资源：

```
condition.notify()
condition.release()
```

## 2.9.4 了解更多

Python对条件同步的实现很有趣。如果没有已经存在的锁传给构造器的话，内部的 `_Condition` 会创建一个 `RLock()` 对象。同时，这个 `RLock` 也会通过 `acquire()` 和 `release()` 管理：

```
class _Condition(_Verbose):
    def __init__(self, lock=None, verbose=None):
        _Verbose.__init__(self, verbose)
        if lock is None:
            lock = RLock()
        self._lock = lock
```

## 2.10 使用事件进行线程同步

事件是线程之间用于通讯的对象。有的线程等待信号，有的线程发出信号。基本上事件对象都会维护一个内部变量，可以通过 `set()` 方法设置为 `true`，也可以通过 `clear()` 方法设置为 `false`。`wait()` 方法将会阻塞线程，直到内部变量为 `true`。

### 2.10.1 如何做...

为了理解通过事件对象实现的线程同步，让我们再一次回到生产者-消费者问题上：

```
# -*- coding: utf-8 -*-

import time
from threading import Thread, Event
import random
items = []
event = Event()

class consumer(Thread):
    def __init__(self, items, event):
        Thread.__init__(self)
        self.items = items
        self.event = event

    def run(self):
        while True:
            time.sleep(2)
            self.event.wait()
            item = self.items.pop()
            print('Consumer notify : %d popped from list by %s' % (item, self.name))
```

```
class producer(Thread):
    def __init__(self, integers, event):
        Thread.__init__(self)
        self.items = items
        self.event = event

    def run(self):
        global item
        for i in range(100):
            time.sleep(2)
            item = random.randint(0, 256)
            self.items.append(item)
            print('Producer notify : item N° %d appended to list by %s' % (item, self.
        ↪name))
            print('Producer notify : event set by %s' % self.name)
            self.event.set()
            print('Produce notify : event cleared by %s '% self.name)
            self.event.clear()

if __name__ == '__main__':
    t1 = producer(items, event)
    t2 = consumer(items, event)
    t1.start()
    t2.start()
    t1.join()
    t2.join()
```

下图是我运行程序时候的运行结果。线程t1在list最后添加值，然后设置event来通知消费者。消费者通过wait()阻塞，直到收到信号的时候从list中取出元素消费。

```

76 *Python Shell*
File Edit Shell Debug Options Windows Help

Producer notify : item 204 appended to list by Thread-1
Producer notify : event set by Thread-1
Produce notify : event cleared by Thread-1
Consumer notify : 204 popped from list by Thread-2

Producer notify : item 98 appended to list by Thread-1
Producer notify : event set by Thread-1
Produce notify : event cleared by Thread-1

Consumer notify : 98 popped from list by Thread-2
Producer notify : item 90 appended to list by Thread-1
Producer notify : event set by Thread-1
Produce notify : event cleared by Thread-1
Consumer notify : 90 popped from list by Thread-2

Producer notify : item 3 appended to list by Thread-1
Producer notify : event set by Thread-1
Produce notify : event cleared by Thread-1
Consumer notify : 3 popped from list by Thread-2

Producer notify : item 162 appended to list by Thread-1
Producer notify : event set by Thread-1
Produce notify : event cleared by Thread-1
Consumer notify : 162 popped from list by Thread-2

Producer notify : item 208 appended to list by Thread-1
Producer notify : event set by Thread-1
Produce notify : event cleared by Thread-1
Consumer notify : 208 popped from list by Thread-2

Producer notify : item 97 appended to list by Thread-1
Producer notify : event set by Thread-1
Produce notify : event cleared by Thread-1
Consumer notify : 97 popped from list by Thread-2

Producer notify : item 233 appended to list by Thread-1
Producer notify : event set by Thread-1
Produce notify : event cleared by Thread-1
Consumer notify : 233 popped from list by Thread-2

```

Ln: 480 Col: 0

## 2.10.2 讨论

producer 类初始化时定义了item的list和 Event , 与条件对象时候的例子不同, 这里的list并不是全局的, 而是通过参数传入的:

```

class consumer(Thread):
    def __init__(self, items, event):
        Thread.__init__(self)
        self.items = items
        self.event = event

```

在run方法中, 每当item创建, producer 类将 newItem添加到list末尾然后发出事件通知。使用事件有两步, 第一步:

```
self.event.set()
```

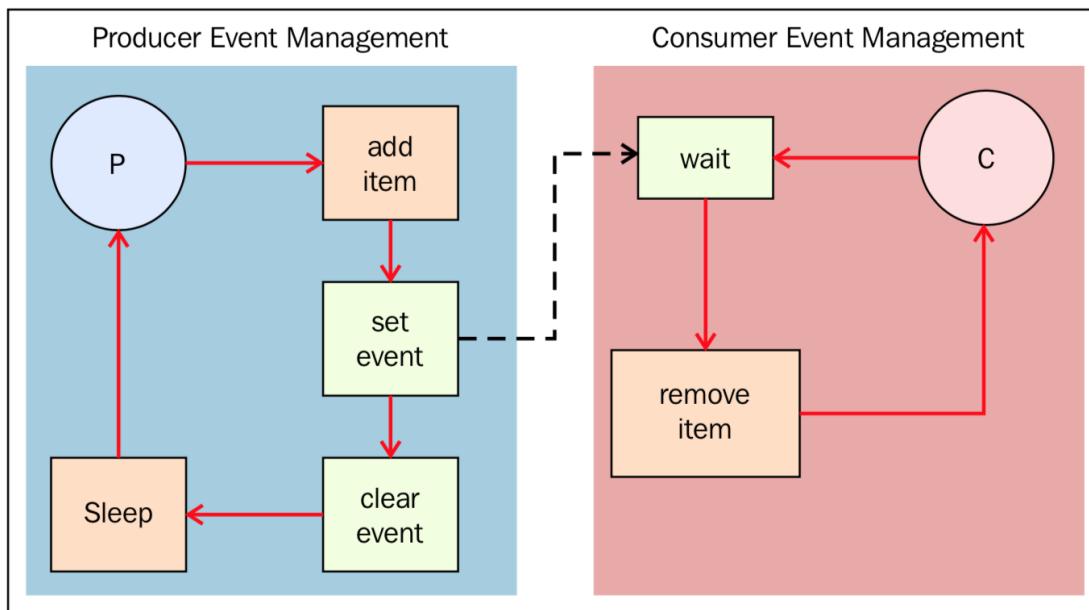
第二步:

```
self.event.clear()
```

consumer 类初始化时也定义了item的list和 Event ()。当item进来的时候，它将其取出:

```
def run(self):
    while True:
        time.sleep(2)
        self.event.wait()
        item = self.items.pop()
        print('Consumer notify : %d popped from list by %s' % (item, self.name))
```

下图可以帮我们认识 producer 和 consumer :



Thread synchronization with event objects

## 2.11 使用with语法

Python从2.5版本开始引入了 with 语法。此语法非常实用，在有两个相关的操作需要在一部分代码块前后分别执行的时候，可以使用 with 语法自动完成。同事，使用 with 语法可以在特定的地方分配和释放资源，因此， with 语法也叫做“上下文管理器”。在threading模块中，所有带有 acquire() 方法和 release() 方法的对象都可以使用上下文管理器。

也就是说，下面的对象可以使用 with 语法:

- Lock
- RLock
- Condition

- Semaphore

### 2.11.1 准备工作

在本节中，我们将使用 with 语法简单地尝试这四个对象。

### 2.11.2 如何做...

下面的例子展示了 with 语法的基本用法，我们有一系列的同步原语，下面尝试用 with 来使用它们：

```
import threading
import logging
logging.basicConfig(level=logging.DEBUG, format='(%(threadName)-10s) %(message)s',)

def threading_with(statement):
    with statement:
        logging.debug('%s acquired via with' % statement)

def threading_not_with(statement):
    statement.acquire()
    try:
        logging.debug('%s acquired directly' % statement)
    finally:
        statement.release()

if __name__ == '__main__':
    # let's create a test battery
    lock = threading.Lock()
    rlock = threading.RLock()
    condition = threading.Condition()
    mutex = threading.Semaphore(1)
    threading_synchronization_list = [lock, rlock, condition, mutex]
    # in the for cycle we call the threading_with e threading_no_with function
    for statement in threading_synchronization_list:
        t1 = threading.Thread(target=threading_with, args=(statement,))
        t2 = threading.Thread(target=threading_not_with, args=(statement,))
        t1.start()
        t2.start()
        t1.join()
        t2.join()
```

下图展示了使用 with 的每一个函数以及用在了什么地方：

```

74 Python Shell
File Edit Shell Debug Options Windows Help
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:55:48) [MSC v.1600 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
(Thread-1) <_thread.lock object at 0x01A29620> acquired via with
(Thread-2) <_thread.lock object at 0x01A29620> acquired directly
(Thread-3) <_thread.RLock owner=3344 count=1> acquired via with
(Thread-4) <_thread.RLock owner=7420 count=1> acquired directly
(Thread-5) <Condition(<_thread.RLock owner=7720 count=1>, 0)> acquired via with
(Thread-6) <Condition(<_thread.RLock owner=6080 count=1>, 0)> acquired directly
(Thread-7) <threading.Semaphore object at 0x01ED8710> acquired via with
(Thread-8) <threading.Semaphore object at 0x01ED8710> acquired directly
>>>

```

Ln: 13 Col: 4

### 2.11.3 讨论

在主程序中，我们定义了一个list，`threading_synchronization_list`，包含要测试的线程同步使用的对象：

```

lock = threading.Lock()
rlock = threading.RLock()
condition = threading.Condition()
mutex = threading.Semaphore(1)
threading_synchronization_list = [lock, rlock, condition, mutex]

```

定义之后，我们可以在 `for` 循环中测试每一个对象：

```

for statement in threading_synchronization_list :
    t1 = threading.Thread(target=threading_with, args=(statement,))
    t2 = threading.Thread(target=threading_not_with, args=(statement,))

```

最后，我们有两个目标函数，其中 `threading_with` 测试了 `with` 语法：

```

def threading_with(statement):
    with statement:
        logging.debug('%s acquired via with' % statement)

```

### 2.11.4 了解更多

在本例中，我们使用了Python的`logging`模块进行输出：

```
logging.basicConfig(level=logging.DEBUG, format='%(threadName)-10s %(message)s',)
```

使用 `% (threadName)` 可以在每次输出的信息都加上线程的名字。`logging` 模块是线程安全的。这样我们可以区分出不同线程的输出。

译者注：译者在博客上写过一篇有关Python的`with`语句的文章，可以参考一下：<https://www.kawabangga.com/posts/2010>

## 2.12 使用 queue 进行线程通信

前面我们已经讨论到，当线程之间如果要共享资源或数据的时候，可能变的非常复杂。如你所见，Python的threading模块提供了很多同步原语，包括信号量，条件变量，事件和锁。如果可以使用这些原语的话，应该优先考虑使用这些，而不是使用queue（队列）模块。队列操作起来更容易，也使多线程编程更安全，因为队列可以将资源的使用通过单线程进行完全控制，并且允许使用更加整洁和可读性更高的设计模式。

Queue常用的方法有以下四个：

- put (): 往queue中放一个item
- get (): 从queue删除一个item，并返回删除的这个item
- task\_done (): 每次item被处理的时候需要调用这个方法
- join (): 所有item都被处理之前一直阻塞

### 2.12.1 如何做...

在本例中，我们将学习如何在threading模块中使用queue。同样，本例中将会有两个实体试图共享临界资源，一个队列。代码如下：

```
from threading import Thread, Event
from queue import Queue
import time
import random
class producer(Thread):
    def __init__(self, queue):
        Thread.__init__(self)
        self.queue = queue

    def run(self) :
        for i in range(10):
            item = random.randint(0, 256)
            self.queue.put(item)
            print('Producer notify: item N° %d appended to queue by %s' % (item, self.name))
            time.sleep(1)

class consumer(Thread):
    def __init__(self, queue):
        Thread.__init__(self)
        self.queue = queue

    def run(self):
        while True:
            item = self.queue.get()
            print('Consumer notify : %d popped from queue by %s' % (item, self.name))
            self.queue.task_done()

if __name__ == '__main__':
    queue = Queue()
    t1 = producer(queue)
    t2 = consumer(queue)
    t3 = consumer(queue)
    t4 = consumer(queue)
    t1.start()
```

```
t2.start()
t3.start()
t4.start()
t1.join()
t2.join()
t3.join()
t4.join()
```

代码的运行结果如下：

```
File Edit Shell Debug Options Windows Help
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:55:48) [MSC v.1600 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
Producer notify : item N° 68 appended to queue by Thread-1
Consumer notify : 68 popped from queue by Thread-2
Producer notify : item N° 101 appended to queue by Thread-1
Consumer notify : 101 popped from queue by Thread-2

Producer notify : item N° 64 appended to queue by Thread-1
Consumer notify : 64 popped from queue by Thread-3

Producer notify : item N° 193 appended to queue by Thread-1
Consumer notify : 193 popped from queue by Thread-4

Producer notify : item N° 234 appended to queue by Thread-1
Consumer notify : 234 popped from queue by Thread-2

Consumer notify : 135 popped from queue by Thread-3
Producer notify : item N° 135 appended to queue by Thread-1

Producer notify : item N° 186 appended to queue by Thread-1
Consumer notify : 186 popped from queue by Thread-4

Producer notify : item N° 135 appended to queue by Thread-1
Consumer notify : 135 popped from queue by Thread-2

Producer notify : item N° 217 appended to queue by Thread-1
Consumer notify : 217 popped from queue by Thread-3

Producer notify : item N° 87 appended to queue by Thread-1
Consumer notify : 87 popped from queue by Thread-4
|
```

## 2.12.2 讨论

首先，我们创建一个生产者类。由于我们使用队列存放数字，所以不需要用来存放数字的list了。

```
class producer(Thread):
    def __init__(self, queue):
        Thread.__init__(self)
        self.queue = queue
```

producer 类生产整数，然后通过一个 for 循环将整数放到队列中：

```
def run(self) :
    for i in range(10):
        item = random.randint(0, 256)
        self.queue.put(item)
```

```
    print('Producer notify: item N° %d appended to queue by %s' % (item, self.  
→name))  
    time.sleep(1)
```

生产者使用 `Queue.put(item [,block[, timeout]])` 来往queue中插入数据。Queue是同步的，在插入数据之前内部有一个内置的锁机制。

可能发生两种情况：

- 如果 `block` 为 `True`，`timeout` 为 `None`（这也是默认的选项，本例中使用默认选项），那么可能会阻塞掉，直到出现可用的位置。如果 `timeout` 是正整数，那么阻塞直到这个时间，就会抛出一个异常。
- 如果 `block` 为 `False`，如果队列有闲置那么会立即插入，否则就立即抛出异常（`timeout` 将会被忽略）。本例中，`put()` 检查队列是否已满，然后调用 `wait()` 开始等待。

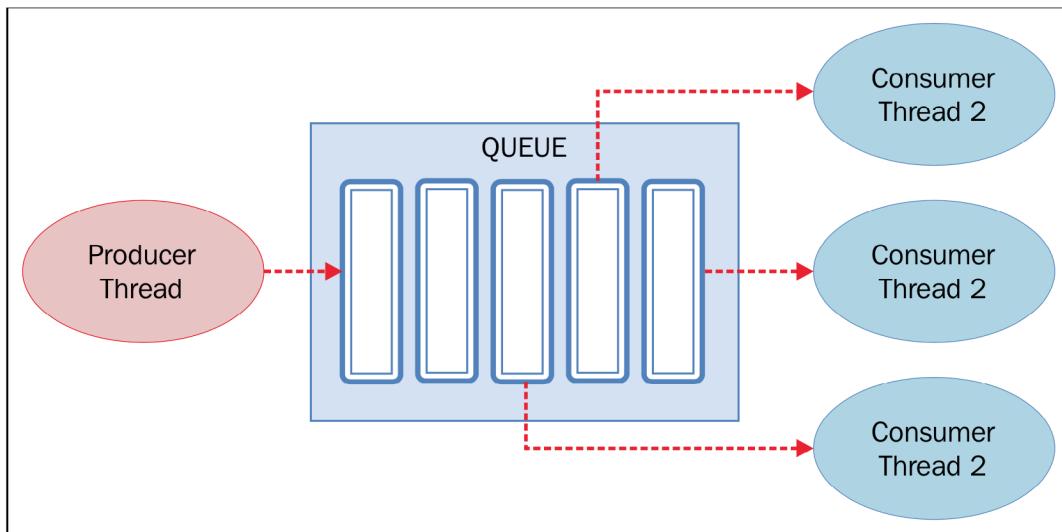
消费者从队列中取出整数然后用 `task_done()` 方法将其标为任务已处理。

消费者使用 `Queue.get([block[, timeout]])` 从队列中取回数据，queue内部也会经过锁的处理。如果队列为空，消费者阻塞。

最后，在主程序中，我们创建线程t作为生产者，t1, t2, t3作为消费者：

```
if __name__ == '__main__':  
    queue = Queue()  
    t1 = producer(queue)  
    t2 = consumer(queue)  
    t3 = consumer(queue)  
    t4 = consumer(queue)  
    t1.start()  
    t2.start()  
    t3.start()  
    t4.start()  
    t1.join()  
    t2.join()  
    t3.join()  
    t4.join()
```

生产者和消费者之间的操作可以用下图来描述：



Thread synchronization with the queue module

## 2.13 评估多线程应用的性能

在本节中，我们将验证GIL的影响，评估多线程应用的性能。前文已经介绍过，GIL是CPython解释器引入的锁，GIL在解释器层面阻止了真正的并行运行。解释器在执行任何线程之前，必须等待当前正在运行的线程释放GIL。事实上，解释器会强迫想要运行的线程必须拿到GIL才能访问解释器的任何资源，例如栈或Python对象等。这也正是GIL的目的——阻止不同的线程并发访问Python对象。这样GIL可以保护解释器的内存，让垃圾回收工作正常。但事实上，这却造成了程序员无法通过并行执行多线程来提高程序的性能。如果我们去掉CPython的GIL，就可以让多线程真正并行执行。GIL并没有影响多处理器并行的线程，只是限制了一个解释器只能有一个线程在运行。

### 2.13.1 如何做...

下面的代码是用来评估多线程应用性能的简单工具。下面的每一个测试都循环调用函数100次，重复执行多次，取速度最快的一次。在 `for` 循环中，我们调用 `non_threaded` 和 `threaded` 函数。同时，我们会不断增加调用次数和线程数来重复执行这个测试。我们会尝试使用1, 2, 3, 4和8线程数来调用线程。在非线程的测试中，我们顺序调用函数与对应线程数一样多的次数。为了保持简单，度量的指标使用Python的内建模块`timer`。

代码如下：

```
from threading import Thread

class threads_object(Thread):
    def run(self):
        function_to_run()

class nothreads_object(object):
    def run(self):
        function_to_run()

def non_threaded(num_iter):
```

```

funcs = []
for i in range(int(num_iter)):
    funcs.append(notthreads_object())
for i in funcs:
    i.run()

def threaded(num_threads):
    funcs = []
    for i in range(int(num_threads)):
        funcs.append(threads_object())
    for i in funcs:
        i.start()
    for i in funcs:
        i.join()

def function_to_run():
    pass

def show_results(func_name, results):
    print("%-23s %4.6f seconds" % (func_name, results))

if __name__ == "__main__":
    import sys
    from timeit import Timer
    repeat = 100
    number = 1
    num_threads = [1, 2, 4, 8]
    print('Starting tests')
    for i in num_threads:
        t = Timer("non_threaded(%s)" % i, "from __main__ import non_threaded")
        best_result = min(t.repeat(repeat=repeat, number=number))
        show_results("non_threaded (%s iters)" % i, best_result)
        t = Timer("threaded(%s)" % i, "from __main__ import threaded")
        best_result = min(t.repeat(repeat=repeat, number=number))
        show_results("threaded (%s threads)" % i, best_result)
    print('Iterations complete')

```

## 2.13.2 讨论

我们一共进行了四次测试(译者注：原文是three，我怀疑原作者不识数，原文的3个线程数也没有写在代码里)，每一次都会使用不同的function进行测试，只要改变function\_to\_run()就可以了。

测试用的机器是 Core 2 Duo CPU – 2.33Ghz。

### 第一次测试

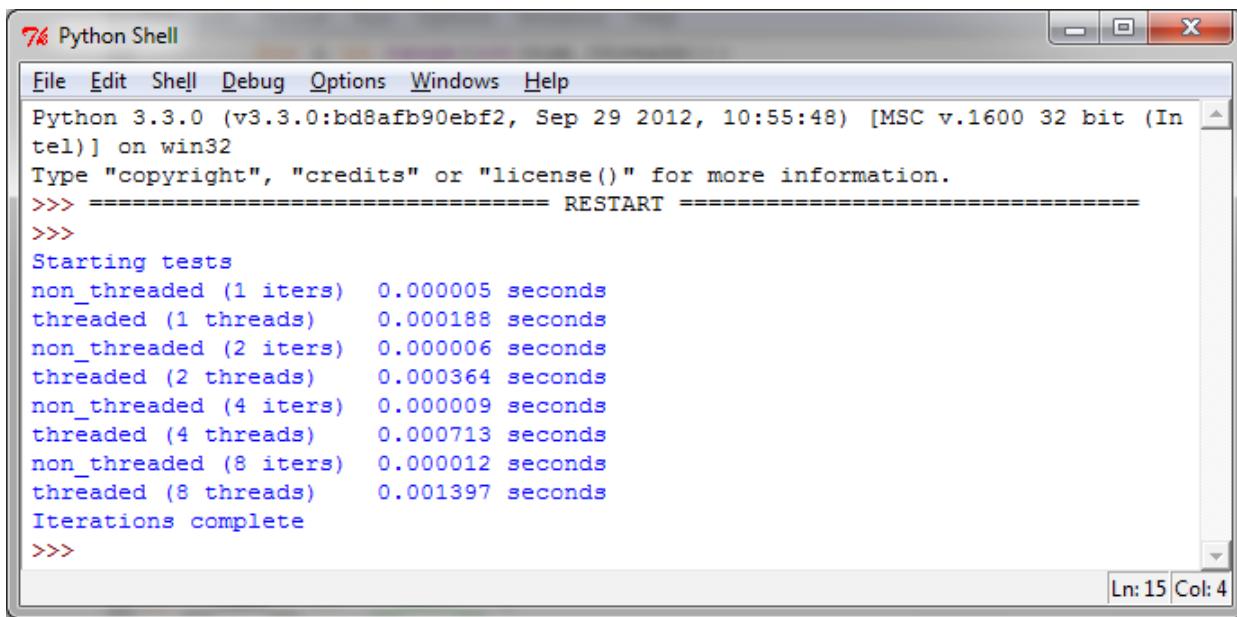
在第一次测试中，我们使用了一个简单的空函数：

```

def function_to_run():
    pass

```

下图展示了我们测试的每个机制的运行速度：



```

74 Python Shell
File Edit Shell Debug Options Windows Help
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:55:48) [MSC v.1600 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
Starting tests
non_threaded (1 iters)  0.000005 seconds
threaded (1 threads)  0.000188 seconds
non_threaded (2 iters)  0.000006 seconds
threaded (2 threads)  0.000364 seconds
non_threaded (4 iters)  0.000009 seconds
threaded (4 threads)  0.000713 seconds
non_threaded (8 iters)  0.000012 seconds
threaded (8 threads)  0.001397 seconds
Iterations complete
>>>
Ln: 15 Col: 4

```

通过结果可以发现，使用线程的开销要比不使用线程的开销大的多。特别的，我们发现随着线程的数量增加，带来的开销是成比例的。4个线程的运行时间是0.0007143秒，8个线程的运行时间是0.001397秒。

## 第二次测试

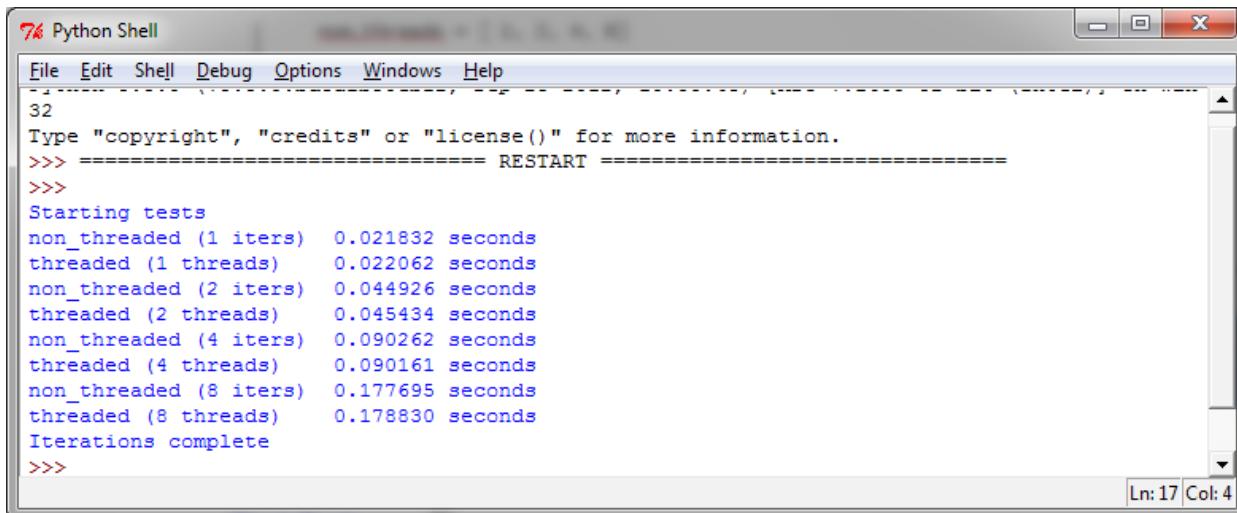
多线程比较常用的一个用途是处理数字，下面的测试计算斐波那契数列，注意这个例子中没有共享的资源，只是测试生成数字数列：

```

def function_to_run():
    a, b = 0, 1
    for i in range(10000):
        a, b = b, a + b

```

输出如下：



```

74 Python Shell
File Edit Shell Debug Options Windows Help
32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
Starting tests
non_threaded (1 iters)  0.021832 seconds
threaded (1 threads)  0.022062 seconds
non_threaded (2 iters)  0.044926 seconds
threaded (2 threads)  0.045434 seconds
non_threaded (4 iters)  0.090262 seconds
threaded (4 threads)  0.090161 seconds
non_threaded (8 iters)  0.177695 seconds
threaded (8 threads)  0.178830 seconds
Iterations complete
>>>
Ln: 17 Col: 4

```

在输出中可以看到，提高线程的数量并没有带来收益。因为**GIL**和线程管理代码的开销，多线程运行永远不可能比函数顺序执行更快。再次提醒一下：**GIL**只允许解释器一次执行一个线程。

### 第三次测试

下面的测试是读1kb的数据1000次，测试用的函数如下：

```
def function_to_run():
    fh=open("C:\\CookBookFileExamples\\test.dat","rb")
    size = 1024
    for i in range(1000):
        fh.read(size)
```

测试的结果如下：

The screenshot shows the Python Shell window with the following output:

```
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:55:48) [MSC v.1600 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
Starting tests
non_threaded (1 iters)  0.074713 seconds
threaded (1 threads)  0.074958 seconds
non_threaded (2 iters)  0.150131 seconds
threaded (2 threads)  0.082746 seconds
non_threaded (4 iters)  0.301600 seconds
threaded (4 threads)  0.168953 seconds
non_threaded (8 iters)  0.604644 seconds
threaded (8 threads)  0.353848 seconds
Iterations complete
>>> |
```

The window has a title bar 'Python Shell' and a status bar at the bottom right indicating 'Ln: 15 Col: 4'.

我们终于看到多线程比非多线程跑的好了的情况了，而且多线程只用了一半的时间。这给我们的启示是，多线程并不是一个标准。一般，我们将会将多线程放入一个队列中，将它们放到一边，执行其他任务。使用多线程执行同一个相同的任务有时候很有用，但用到的时候很少，除非需要大量处理数据输入。

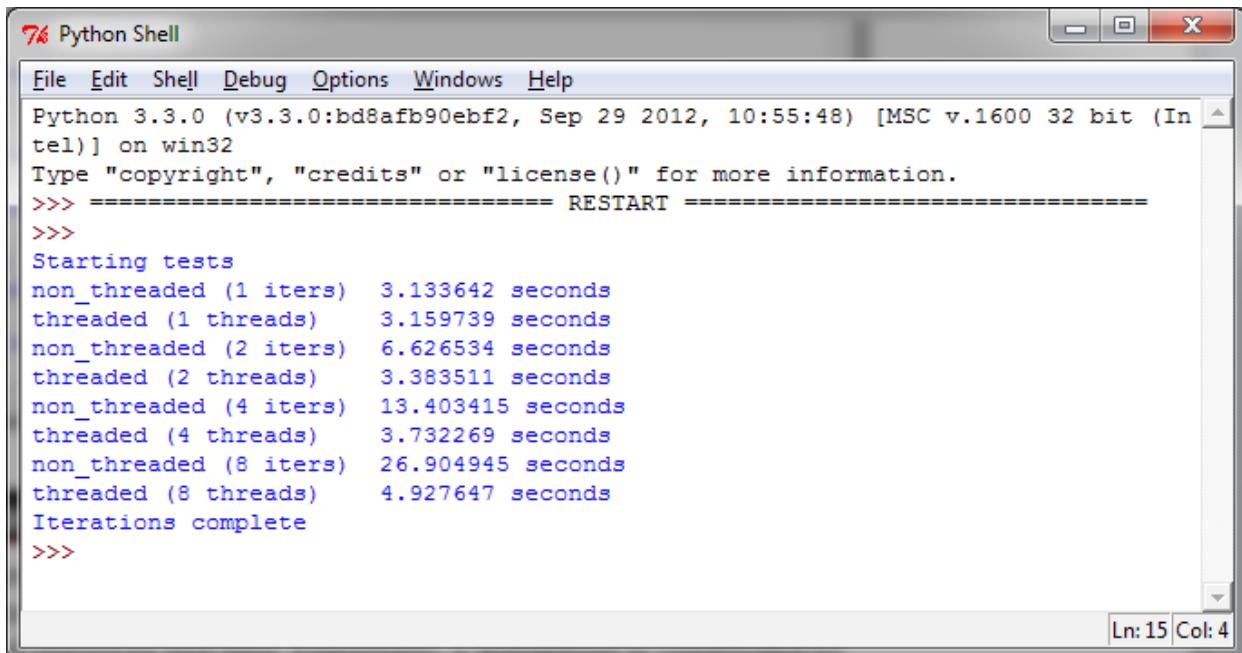
### 第四次测试

在最后的测试中，我们使用 `urllib.request` 测试，这是一个Python模块，可以发送URL请求。此模块基于 `socket`，使用C语言编写并且是线程安全的。

下面的代码尝试读取 `https://www.packtpub.com` 的主页并且读取前1k的数据：

```
def function_to_run():
    import urllib.request
    for i in range(10):
        with urllib.request.urlopen("https://www.packtpub.com/") as f:
            f.read(1024)
```

运行结果如下：



The screenshot shows a Windows-style Python Shell window titled "Python Shell". The window has a menu bar with File, Edit, Shell, Debug, Options, Windows, Help. The main area displays the following text:

```
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:55:48) [MSC v.1600 32 bit (In tel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
Starting tests
non_threaded (1 iters) 3.133642 seconds
threaded (1 threads) 3.159739 seconds
non_threaded (2 iters) 6.626534 seconds
threaded (2 threads) 3.383511 seconds
non_threaded (4 iters) 13.403415 seconds
threaded (4 threads) 3.732269 seconds
non_threaded (8 iters) 26.904945 seconds
threaded (8 threads) 4.927647 seconds
Iterations complete
>>>
```

In the bottom right corner of the window, there is a status bar with "Ln: 15 Col: 4".

可以看到，在 I/O 期间，GIL 释放了。多线程执行比单线程快的多。鉴于大多数应用需要很多I/O 操作，GIL 并没有限制程序员在这方面使用多线程对程序进行性能优化。

### 2.13.3 了解更多

你应该记住，增加线程并不会提高应用启动的时间，但是可以支持并发。例如，一次性创建一个线程池，并重用worker会很有用。这可以让我们切分一个大的数据集，用同样的函数处理不同的部分（生产者消费者模型）。上面这些测试并不是并发应用的模型，只是尽量简单的测试。那么GIL会成为试图发挥多线程应用潜能的纯Python开发的瓶颈吗？是的。线程是编程语言的架构，CPython解释器是线程和操作系统的桥梁。这就是为什么Jython, IronPython没有GIL的原因（译者注：Pypy也没有），因为它不是必要的。



# CHAPTER 3

---

## 第三章 基于进程的并行

---

### 3.1 介绍

在之前的章节中，我们见识了如何用线程实现并发的应用。本章节将会介绍基于进程的并行。本章的重点将会集中在Python的multiprocessing和mpi4py这两个模块上。

multiprocessing是Python标准库中的模块，实现了共享内存机制，也就是说，可以让运行在不同处理器核心的进程能读取共享内存。

mpi4py库实现了消息传递的编程范例（设计模式）。简单来说，就是进程之间不靠任何共享信息来进行通讯（也叫做shared nothing），所有的交流都通过传递信息代替。

这方面与使用共享内存通讯，通过加锁或类似机制实现互斥的技术行成对比。在信息传递的代码中，进程通过send()和receive进行交流。

在Python多进程的官方文档中，明确指出multiprocessing模块要求，使用此模块的函数的main模块对于类来说必须是可导入的（<https://docs.python.org/3.3/library/multiprocessing.html>）。

\_\_main\_\_在IELE中并不是可以导入的，即使你在IDLE中将文件当做一个脚本来运行。为了能正确使用此模块，本章我们将在命令行使用下面的命令运行脚本：

```
python multiprocessing example.py
```

这里，multiprocessing\_example.py是脚本的文件名。本章使用的解释器是Python3.3（实际上使用Python2.7也是可以的）。

（译者注，抱歉，这段译者无法理解原文和Python文档的意思。不过通过实验，我发现要多进程运行一个函数，这个函数必须从外部文件导入。比如说下面这样就不行：

```
>>> from multiprocessing import Pool
>>> p = Pool(5)
>>> def f(x):
...     return x*x
...
>>> p.map(f, [1,2,3])
Process PoolWorker-1:
```

```

Process PoolWorker-2:
Process PoolWorker-3:
Traceback (most recent call last):
AttributeError: 'module' object has no attribute 'f'
AttributeError: 'module' object has no attribute 'f'
AttributeError: 'module' object has no attribute 'f'

```

上面脚本来自Python官网文档。如果稍作修改，将需要多进程运行的目标函数放到文件里导入运行，就没有问题了：

```

In [1]: from multiprocessing import Pool
In [2]: p = Pool(5)
In [4]: import func
In [5]: p.map(func.f, [1, 2, 3])
Out[5]: [1, 4, 9]

```

译者注结束)

## 3.2 如何产生一个进程

“产生”（spawn）的意思是，由父进程创建子进程。父进程既可以在产生子进程之后继续异步执行，也可以暂停等待子进程创建完成之后再继续执行。Python的multiprocessing库通过以下几步创建进程：

1. 创建进程对象
2. 调用 `start()` 方法，开启进程的活动
3. 调用 `join()` 方法，在进程结束之前一直等待

### 3.2.1 如何做...

下面的例子创建了5个进程，每一个进程都分配了 `foo(i)` 函数，`i` 表示进程的id：

```

# -*- coding: utf-8 -*-
import multiprocessing

def foo(i):
    print ('called function in process: %s' %i)
    return

if __name__ == '__main__':
    Process_jobs = []
    for i in range(5):
        p = multiprocessing.Process(target=foo, args=(i,))
        Process_jobs.append(p)
        p.start()
        p.join()

```

执行本例需要打开命令行，到文件 `spawn_a_process.py`（脚本名字）所在的目录下，然后输入下面的命令执行：

```
python spawn_a_process.py
```

我们会得到以下结果:

```
$ python process_2.py
called function in process: 0
called function in process: 1
called function in process: 2
called function in process: 3
called function in process: 4
```

## 3.2.2 讨论

按照本节前面提到的步骤，创建进程对象首先需要引入multiprocessing模块:

```
import multiprocessing
```

然后，我们在主程序中创建进程对象:

```
p = multiprocessing.Process(target=foo, args=(i,))
```

最后，我们调用 start() 方法启动:

```
p.start()
```

进程对象的时候需要分配一个函数，作为进程的执行任务，本例中，这个函数是 foo()。我们可以用元组的形式给函数传递一些参数。最后，使用进程对象调用 join() 方法。

如果没有 join()，主进程退出之后子进程会留在idle中，你必须手动杀死它们。

## 3.2.3 了解更多

这是因为，子进程创建的时候需要导入包含目标函数的脚本。通过在 \_\_main\_\_ 代码块中实例化进程对象，我们可以预防无限递归调用。最佳实践是在不同的脚本文件中定义目标函数，然后导入进来使用。所以上面的代码可以修改为:

```
import multiprocessing
import target_function
if __name__ == '__main__':
    Process_jobs = []
    for i in range(5):
        p = multiprocessing.Process(target=target_function.function, args=(i,))
        Process_jobs.append(p)
        p.start()
        p.join()
```

target\_function.py 的内容如下:

```
def function(i):
    print('called function in process: %s' %i)
    return
```

输出和上面一样。

## 3.3 如何为一个进程命名

在上一节的例子中，我们创建了一个进程，并为其分配了目标函数和函数变量。然而如果能给进程分配一个名字，那么debug的时候就更方便了。

### 3.3.1 如何做...

命名进程的方法和前一章中介绍的命名线程差不多（参考第二章，基于线程的并行，第四节，如何确定当前的线程）。

下面的代码在主程序中创建了一个有名字的进程和一个没有名字的进程，目标函数都是 `foo()` 函数。

```
# 命名一个进程
import multiprocessing
import time

def foo():
    name = multiprocessing.current_process().name
    print("Starting %s \n" % name)
    time.sleep(3)
    print("Exiting %s \n" % name)

if __name__ == '__main__':
    process_with_name = multiprocessing.Process(name='foo_process', target=foo)
    process_with_name.daemon = True # 注意原代码有这一行，但是译者发现删掉这一行才能得到正确输出
    process_with_default_name = multiprocessing.Process(target=foo)
    process_with_name.start()
    process_with_default_name.start()
```

运行上面的代码，打开终端输入：

```
python naming_process.py
```

输出的结果如下：

```
$ python naming_process.py
Starting foo_process
Starting Process-2
Exiting foo_process
Exiting Process-2
```

### 3.3.2 讨论

这个过程明明线程很像。明明进程需要为进程对象提供 `name` 参数：

```
process_with_name = multiprocessing.Process(name='foo_process', target=foo)
```

在本例子中，进程的名字就是 `foo_function`。如果子进程需要知道父进程的名字，可以使用以下声明：

```
name = multiprocessing.current_process().name
```

然后就能看见父进程的名字。

## 3.4 如何在后台运行一个进程

如果需要处理比较巨大的任务，又不需要人为干预，将其作为后台进程执行是个非常常用的编程模型。此进程又可以和其他进程并发执行。通过Python的multiprocessing模块的后台进程选项，我们可以让进程在后台运行。

### 3.4.1 如何做...

运行后台进程，可以参照以下代码：

```
import multiprocessing
import time

def foo():
    name = multiprocessing.current_process().name
    print("Starting %s" % name)
    time.sleep(3)
    print("Exiting %s" % name)

if __name__ == '__main__':
    background_process = multiprocessing.Process(name='background_process', target=foo)
    background_process.daemon = True
    NO_background_process = multiprocessing.Process(name='NO_background_process', target=foo)
    NO_background_process.daemon = False
    background_process.start()
    NO_background_process.start()
```

运行上面的脚本，需要使用下面的命令：

```
python background_process.py
```

最后的输出如下：

```
$ python background_process.py
Starting NO_background_process
Exiting NO_background_process
```

### 3.4.2 讨论

为了在后台运行进程，我们设置 daemon 参数为 True

```
background_process.daemon = True
```

在非后台运行的进程会看到一个输出，后台运行的没有输出，后台运行进程在主进程结束之后会自动结束。

### 3.4.3 了解更多

注意，后台进程不允许创建子进程。否则，当后台进程跟随父进程退出的时候，子进程会变成孤儿进程。另外，它们并不是Unix的守护进程或服务（daemons or services），所以当非后台进程退出，它们会被终结。

## 3.5 如何杀掉一个进程

我们可以使用 `terminate()` 方法立即杀死一个进程。另外，我们可以使用 `is_alive()` 方法来判断一个进程是否还存活。

### 3.5.1 如何做...

在本例中，创建一个目标函数为 `foo()` 的进程。启动之后，我们通过 `terminate()` 方法杀死它。

```
# 杀死一个进程
import multiprocessing
import time

def foo():
    print('Starting function')
    time.sleep(0.1)
    print('Finished function')

if __name__ == '__main__':
    p = multiprocessing.Process(target=foo)
    print('Process before execution:', p, p.is_alive())
    p.start()
    print('Process running:', p, p.is_alive())
    p.terminate()
    print('Process terminated:', p, p.is_alive())
    p.join()
    print('Process joined:', p, p.is_alive())
    print('Process exit code:', p.exitcode)
```

输出如下：

```
76 Python Shell
File Edit Shell Debug Options Windows Help
Python 3.3.0 (v3.3.0:bd8af890ebf2, Sep 29 2012, 10:55:48) [MSC v.1600 32 bit (Int
el)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
Process before execution: <Process(Process-1, initial)> False
Process running: <Process(Process-1, started)> True
Process terminated: <Process(Process-1, stopped[SIGTERM])> True
Process joined: <Process(Process-1, stopped[SIGTERM])> False
Process exit code: -15
>>> |
```

### 3.5.2 讨论

我们创建了一个线程，然后用 `is_alive()` 方法监控它的声明周期。然后通过调用 `terminate()` 方法结束进程。

最后，我们通过读进程的 `ExitCode` 状态码（`status code`）验证进程已经结束，`ExitCode` 可能的值如下：

- == 0: 没有错误正常退出

- > 0: 进程有错误，并以此状态码退出
- < 0: 进程被 -1 \* 的信号杀死并以此作为 ExitCode 退出

在我们的例子中，输出的 ExitCode 是 -15。负数表示子进程被数字为15的信号杀死。

## 3.6 如何在子类中使用进程

实现一个自定义的进程子类，需要以下三步：

- 定义 Process 的子类
- 覆盖 `__init__(self [,args])` 方法来添加额外的参数
- 覆盖 `run(self, [.args])` 方法来实现 Process 启动的时候执行的任务

创建 Process 子类之后，你可以创建它的实例并通过 `start()` 方法启动它，启动之后会运行 `run()` 方法。

### 3.6.1 如何做...

我们将使用子类的形式重写之前的例子：

```
# -*- coding: utf-8 -*-
# 自定义子类进程
import multiprocessing

class MyProcess(multiprocessing.Process):
    def run(self):
        print ('called run method in process: %s' % self.name)
        return

if __name__ == '__main__':
    jobs = []
    for i in range(5):
        p = MyProcess()
        jobs.append(p)
        p.start()
        p.join()
```

输入以下命令运行脚本：

```
python subclass_process.py
```

运行结果如下：

```
$ python subclass.py
called run method in process: MyProcess-1
called run method in process: MyProcess-2
called run method in process: MyProcess-3
called run method in process: MyProcess-4
called run method in process: MyProcess-5
```

## 3.6.2 讨论

每一个继承了 `Process` 并重写了 `run()` 方法的子类都代表一个进程。此方法是进程的入口：

```
class MyProcess(multiprocessing.Process):
    def run(self):
        print ('called run method in process: %s' % self.name)
        return
```

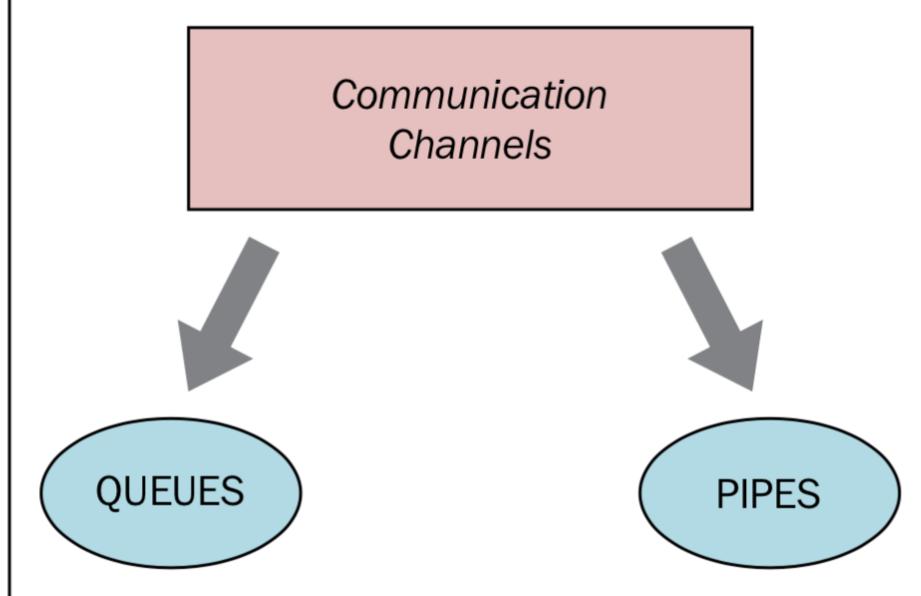
在主程序中，我们创建了一些 `MyProcess()` 的子类。当 `start()` 方法被调用的时候进程开始执行：::

`p = MyProcess()` `p.start()`

`join()` 命令可以让主进程等待其他进程结束最后退出。

## 3.7 如何在进程之间交换对象

并行应用常常需要在进程之间交换数据。Multiprocessing库有两个Communication Channel可以交换对象：队列(queue)和管道 (pipe) 。



Communication channels in the multiprocessing module

### 3.7.1 使用队列交换对象

我们可以通过队列数据结构来共享对象。

`Queue` 返回一个进程共享的队列，是线程安全的，也是进程安全的。任何可序列化的对象（Python通过 `pickable` 模块序列化对象）都可以通过它进行交换。

### 3.7.2 如何做...

在下面的例子中，我们将展示如何使用队列来实现生产者-消费者问题。Producer 类生产item放到队列中，然后 Consumer 类从队列中移除它们。

```
import multiprocessing
import random
import time

class Producer(multiprocessing.Process):
    def __init__(self, queue):
        multiprocessing.Process.__init__(self)
        self.queue = queue

    def run(self):
        for i in range(10):
            item = random.randint(0, 256)
            self.queue.put(item)
            print("Process Producer : item %d appended to queue %s" % (item, self.
        ↪name))
            time.sleep(1)
            print("The size of queue is %s" % self.queue.qsize())

class Consumer(multiprocessing.Process):
    def __init__(self, queue):
        multiprocessing.Process.__init__(self)
        self.queue = queue

    def run(self):
        while True:
            if self.queue.empty():
                print("the queue is empty")
                break
            else:
                time.sleep(2)
                item = self.queue.get()
                print('Process Consumer : item %d popped from by %s \n' % (item, self.
        ↪name))
                time.sleep(1)

if __name__ == '__main__':
    queue = multiprocessing.Queue()
    process_producer = Producer(queue)
    process_consumer = Consumer(queue)
    process_producer.start()
    process_consumer.start()
    process_producer.join()
    process_consumer.join()
```

运行结果如下（译者注：macOS High Sierra运行失败，错误是 NotImplementedError 可能是因为 self.\_sem.\_semlock.\_get\_value() 没有实现）：

```
C:\Python CookBook\Chapter 3 - Process Based Parallelism\Example Codes
Chapter 3>python using_queue.py
Process Producer : item 69 appended to queue producer-1
The size of queue is 1
Process Producer : item 168 appended to queue producer-1
The size of queue is 2
```

```

Process Consumer : item 69 popped from by consumer-2
Process Producer : item 235 appended to queue producer-1
The size of queue is 2
Process Producer : item 152 appended to queue producer-1
The size of queue is 3
Process Producer : item 213 appended to queue producer-1
Process Consumer : item 168 popped from by consumer-2
The size of queue is 3
Process Producer : item 35 appended to queue producer-1
The size of queue is 4
Process Producer : item 218 appended to queue producer-1
The size of queue is 5
Process Producer : item 175 appended to queue producer-1
Process Consumer : item 235 popped from by consumer-2
The size of queue is 5
Process Producer : item 140 appended to queue producer-1
The size of queue is 6
Process Producer : item 241 appended to queue producer-1
The size of queue is 7
Process Consumer : item 152 popped from by consumer-2
Process Consumer : item 213 popped from by consumer-2
Process Consumer : item 35 popped from by consumer-2
Process Consumer : item 218 popped from by consumer-2
Process Consumer : item 175 popped from by consumer-2
Process Consumer : item 140 popped from by consumer-2
Process Consumer : item 241 popped from by consumer-2
the queue is empty

```

### 3.7.3 如何做...

我们使用 multiprocessing 类在主程序中创建了 Queue 的实例:

```

if __name__ == '__main__':
    queue = multiprocessing.Queue()

```

然后我们创建了两个进程，生产者和消费者， Queue 对象作为一个属性。

```

process_producer = Producer(queue)
process_consumer = Consumer(queue)

```

生产者类负责使用 put() 方法放入10个item:

```

for i in range(10):
    item = random.randint(0, 256)
    self.queue.put(item)

```

消费者进程负责使用 get() 方法从队列中移除item，并且确认队列是否为空，如果为空，就执行 break 跳出 while 循环:

```

def run(self):
    while True:
        if self.queue.empty():
            print("the queue is empty")
            break
        else:
            time.sleep(2)

```

```

        item = self.queue.get()
        print('Process Consumer : item %d popped from by %s \n' % (item, self.
    ↪name))
        time.sleep(1)

```

### 3.7.4 了解更多

队列还有一个 `JoinableQueue` 子类，它有以下两个额外的方法：

- `task_done()`: 此方法意味着之前入队的一个任务已经完成，比如，`get()` 方法从队列取回 `item` 之后调用。所以此方法只能被队列的消费者调用。
- `join()`: 此方法将进程阻塞，直到队列中的 `item` 全部被取出并执行。

( Microndgt 注：因为使用队列进行通信是一个单向的，不确定的过程，所以你不知道什么时候队列的元素被取出来了，所以使用者者 `task_done` 来表示已经队列里的一个任务已经完成。

这个方法一般和 `join` 一起使用，当队列的所有任务都处理之后，也就是说 `put` 到队列的每个任务都调用了 `task_done` 方法后，`join` 才会完成阻塞。)

### 3.7.5 使用管道交换对象

第二种 Communication Channel 是管道。

一个管道可以做以下事情：

- 返回一对被管道连接的连接对象
- 然后对象就有了 `send/receive` 方法可以在进程之间通信

### 3.7.6 如何做...

下面是管道用法的一个简单示例。这里有一个进程管道从0到9发出数字，另一个进程接收数字并进行平方计算。

```

import multiprocessing

def create_items(pipe):
    output_pipe, _ = pipe
    for item in range(10):
        output_pipe.send(item)
    output_pipe.close()

def multiply_items(pipe_1, pipe_2):
    close, input_pipe = pipe_1
    close.close()
    output_pipe, _ = pipe_2
    try:
        while True:
            item = input_pipe.recv()
            output_pipe.send(item * item)
    except EOFError:
        output_pipe.close()

if __name__ == '__main__':

```

```

# 第一个进程管道发出数字
pipe_1 = multiprocessing.Pipe(True)
process_pipe_1 = multiprocessing.Process(target=create_items, args=(pipe_1,))
process_pipe_1.start()
# 第二个进程管道接收数字并计算
pipe_2 = multiprocessing.Pipe(True)
process_pipe_2 = multiprocessing.Process(target=multiply_items, args=(pipe_1,_
pipe_2,))
process_pipe_2.start()
pipe_1[0].close()
pipe_2[0].close()
try:
    while True:
        print(pipe_2[1].recv())
except EOFError:
    print("End")

```

程序的输出如下：

```

Python Shell
File Edit Shell Debug Options Windows Help
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:55:48) [MSC v.1600 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
0
1
4
9
16
25
36
49
64
81
End
>>> |

```

### 3.7.7 讨论

`Pipe()` 函数返回一对通过双向管道连接起来的对象。在本例中，`out_pipe` 包含数字0-9，通过目标函数 `create_items()` 产生：

```

def create_items(pipe):
    output_pipe, _ = pipe
    for item in range(10):
        output_pipe.send(item)
    output_pipe.close()

```

在第二个进程中，我们有两个管道，输入管道和包含结果的输出管道：

```

process_pipe_2 = multiprocessing.Process(target=multiply_items, args=(pipe_1, pipe_2,
))

```

最后打印出结果：

```

try:
    while True:
        print(pipe_2[1].recv())
except EOFError:
    print("End")

```

## 3.8 进程如何同步

多个进程可以协同工作来完成一项任务。通常需要共享数据。所以在多进程之间保持数据的一致性就很重要了。需要共享数据协同的进程必须以适当的策略来读写数据。相关的同步原语和线程的库很类似。

进程的同步原语如下：

- **Lock**: 这个对象可以有两种装填：锁住的（locked）和没锁住的（unlocked）。一个Lock对象有两个方法，`acquire()` 和 `release()`，来控制共享数据的读写权限。
- **Event**: 实现了进程间的简单通讯，一个进程发事件的信号，另一个进程等待事件的信号。Event 对象有两个方法，`set()` 和 `clear()`，来管理自己内部的变量。
- **Condition**: 此对象用来同步部分工作流程，在并行的进程中，有两个基本的方法：`wait()` 用来等待进程，`notify_all()` 用来通知所有等待此条件的进程。
- **Semaphore**: 用来共享资源，例如，支持固定数量的共享连接。
- **Rlock**: 递归锁对象。其用途和方法同 Threading 模块一样。
- **Barrier**: 将程序分成几个阶段，适用于有些进程必须在某些特定进程之后执行。处于障碍（Barrier）之后的代码不能同处于障碍之前的代码并行。

### 3.8.1 如何做...

下面的代码展示了如何使用 `barrier()` 函数来同步两个进程。我们有4个进程，进程1和进程2由barrier语句管理，进程3和进程4没有同步策略。

```

import multiprocessing
from multiprocessing import Barrier, Lock, Process
from time import time
from datetime import datetime

def test_with_barrier(synchronizer, serializer):
    name = multiprocessing.current_process().name
    synchronizer.wait()
    now = time()
    with serializer:
        print("process %s ----> %s" % (name, datetime.fromtimestamp(now)))

def test_without_barrier():
    name = multiprocessing.current_process().name
    now = time()
    print("process %s ----> %s" % (name, datetime.fromtimestamp(now)))

if __name__ == '__main__':
    synchronizer = Barrier(2)
    serializer = Lock()
    Process(name='p1 - test_with_barrier', target=test_with_barrier,
    ↴args=(synchronizer, serializer)).start()

```

```

Process(name='p2 - test_with_barrier', target=test_with_barrier,_
→args=(synchronizer,serializer)).start()
Process(name='p3 - test_without_barrier', target=test_without_barrier).start()
Process(name='p4 - test_without_barrier', target=test_without_barrier).start()

```

运行下面的代码，将看到进程1和进程2在同一时间打印：

```

$ python process_barrier.py
process p1 - test_with_barrier ----> 2015-05-09 11:11:33.291229
process p2 - test_with_barrier ----> 2015-05-09 11:11:33.291229
process p3 - test_without_barrier ----> 2015-05-09 11:11:33.310230
process p4 - test_without_barrier ----> 2015-05-09 11:11:33.333231

```

(译者注：译者在实际运行了10次，没有一次时间是相同的，感觉这个地方同一时间打印出来的影响因素很多。只能看到 with\_barrier 的进程1和2比 without\_barrier 的进程3和4时间差的小很多。)

### 3.8.2 讨论

在主程序中，我们创建了四个进程，然后我们需要一个锁和一个barrier来进程同步。barrier声明的第二个参数代表要管理的进程总数：

```

if __name__ == '__main__':
    synchronizer = Barrier(2)
    serializer = Lock()
    Process(name='p1 - test_with_barrier', target=test_with_barrier,_
→args=(synchronizer,serializer)).start()
    Process(name='p2 - test_with_barrier', target=test_with_barrier,_
→args=(synchronizer,serializer)).start()
    Process(name='p3 - test_without_barrier', target=test_without_barrier).start()
    Process(name='p4 - test_without_barrier', target=test_without_barrier).start()

```

test\_with\_barrier 函数调用了 barrier 的 wait() 方法：

```

def test_with_barrier(synchronizer, serializer):
    name = multiprocessing.current_process().name
    synchronizer.wait()

```

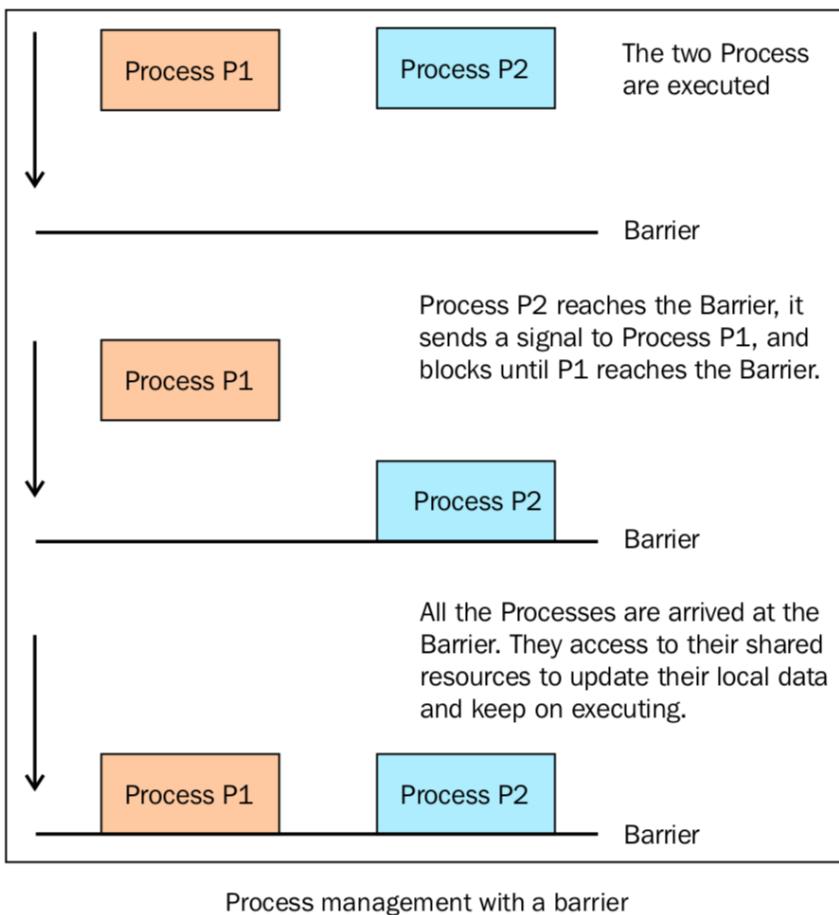
当两个进程都调用 wait() 方法的时候，它们会一起继续执行：

```

now = time()
with serializer:
    print("process %s ----> %s" % (name, datetime.fromtimestamp(now)))

```

下面这幅图表示了 barrier 如何同步两个进程：



## 3.9 如何在进程之间管理状态

Python的多进程模块提供了在所有的用户间管理共享信息的管理者(Manager)。一个管理者对象控制着持有Python对象的服务进程，并允许其它进程操作共享对象。

管理者有以下特性：

- 它控制着管理共享对象的服务进程
- 它确保当某一进程修改了共享对象之后，所有的进程拿到额共享对象都得到了更新

### 3.9.1 如何做...

下面来看一个再进程之间共享对象的例子：

1. 首先，程序创建了一个管理者的字典，在 n 个 taskWorkers 之间共享，每个worker更新字典的某一个index。
2. 所有的worker完成之后，新的列表打印到 stdout：

```

import multiprocessing

def worker(dictionary, key, item):
    dictionary[key] = item
    print("key = %d value = %d" % (key, item))

if __name__ == '__main__':
    mgr = multiprocessing.Manager()
    dictionary = mgr.dict()
    jobs = [multiprocessing.Process(target=worker, args=(dictionary, i, i*2)) for i in range(10)]
    for j in jobs:
        j.start()
    for j in jobs:
        j.join()
    print('Results:', dictionary)

```

(译者注: 源代码中少了一行print, 译者加上了, 能得到和书中输出一样的结果)

运行结果如下:

```

$ python manager.py
key = 0 value = 0
key = 3 value = 6
key = 2 value = 4
key = 1 value = 2
key = 4 value = 8
key = 5 value = 10
key = 8 value = 16
key = 6 value = 12
key = 7 value = 14
key = 9 value = 18
Results: {0: 0, 3: 6, 2: 4, 1: 2, 4: 8, 5: 10, 8: 16, 6: 12, 7: 14, 9: 18}

```

## 3.9.2 讨论

我们在先声明了一个manager:

```
mgr = multiprocessing.Manager()
```

下面一行创建了 dictionary 类型的一个数据结构:

```
dictionary = mgr.dict()
```

然后, 启动多进程:

```

jobs = [multiprocessing.Process(target=worker, args=(dictionary, i, i*2)) for i in range(10)]
for j in jobs:
    j.start()

```

这里, 目标函数 taskWorker 往字典中添加一个item:

```

def worker(dictionary, key, item):
    dictionary[key] = item

```

最后，我们得到字典所有的值并打印出来：

```
for j in jobs:
    j.join()
print('Results:', dictionary)
```

## 3.10 如何使用进程池

多进程库提供了 Pool 类来实现简单的多进程任务。 Pool 类有以下方法：

- apply (): 直到得到结果之前一直阻塞。
- apply\_async (): 这是 apply () 方法的一个变体，返回的是一个result对象。这是一个异步的操作，在所有的子类执行之前不会锁住主进程。
- map (): 这是内置的 map () 函数的并行版本。在得到结果之前一直阻塞，此方法将可迭代的数据的每一个元素作为进程池的一个任务来执行。
- map\_async (): 这是 map () 方法的一个变体，返回一个result对象。如果指定了回调函数，回调函数应该是 callable 的，并且只接受一个参数。当 result 准备好时会自动调用回调函数（除非调用失败）。回调函数应该立即完成，否则，持有 result 的进程将被阻塞。

### 3.10.1 如何做...

下面的例子展示了如果通过进程池来执行一个并行应用。我们创建了有4个进程的进程池，然后使用 map () 方法进行一个简单的计算。

```
import multiprocessing

def function_square(data):
    result = data*data
    return result

if __name__ == '__main__':
    inputs = list(range(100))
    pool = multiprocessing.Pool(processes=4)
    pool_outputs = pool.map(function_square, inputs)
    pool.close()
    pool.join()
    print ('Pool      :', pool_outputs)
```

计算的结果如下：

```
$ python poll.py
('Pool      :', [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256,
  ↪ 289, 324, 361, 400, 441, 484, 529, 576, 625, 676, 729, 784, 841, 900, 961, 1024,
  ↪ 1089, 1156, 1225, 1296, 1369, 1444, 1521, 1600, 1681, 1764, 1849, 1936, 2025, 2116,
  ↪ 2209, 2304, 2401, 2500, 2601, 2704, 2809, 2916, 3025, 3136, 3249, 3364, 3481, 3600,
  ↪ 3721, 3844, 3969, 4096, 4225, 4356, 4489, 4624, 4761, 4900, 5041, 5184, 5329, 5476,
  ↪ 5625, 5776, 5929, 6084, 6241, 6400, 6561, 6724, 6889, 7056, 7225, 7396, 7569, 7744,
  ↪ 7921, 8100, 8281, 8464, 8649, 8836, 9025, 9216, 9409, 9604, 9801])
```

### 3.10.2 如何做...

`multiprocessing.Pool` 方法在输入元素上应用 `function_square` 方法来执行简单的计算。并行的进程数量是4:

```
pool = multiprocessing.Pool(processes=4)
```

`pool.map` 方法将一些独立的任务提交给进程池:

```
pool_outputs = pool.map(function_square, inputs)
```

`inputs` 是一个从 0 到 100 的list:

```
inputs = list(range(100))
```

计算的结果存储在 `pool_outputs` 中。最后的结果打印出来:

```
print ('Pool      :', pool_outputs)
```

需要注意的是，`pool.map()` 方法的记过和Python内置的 `map()` 结果是相同的，不同的是 `pool.map()` 是通过多个并行进程计算的。

## 3.11 使用Python的mpi4py模块

Python 提供了很多MPI模块写并行程序。其中 `mpi4py` 是一个又有意思的库。它在MPI-1/2顶层构建，提供了面向对象的接口，紧跟C++绑定的 MPI-2。MPI的C语言用户可以无需学习新的接口就可以上手这个库。所以，它成为了Python中最广泛使用的MPI库。

此模块包含的主要应用有：

- 点对点通讯
- 集体通讯
- 拓扑

### 3.11.1 准备工作

在Windows中安装 `mpi4py` 的过程如下（其他操作系统可以参考 <http://mpi4py.scipy.org/docs/usrman/install.html>）：

1. 从MPI软件库(<http://www.mpich.org/downloads/>)下载 `mpich`。

**MPICH**

High-Performance Portable MPI

Home About Downloads Documentation Support ABI Compatibility Initiative

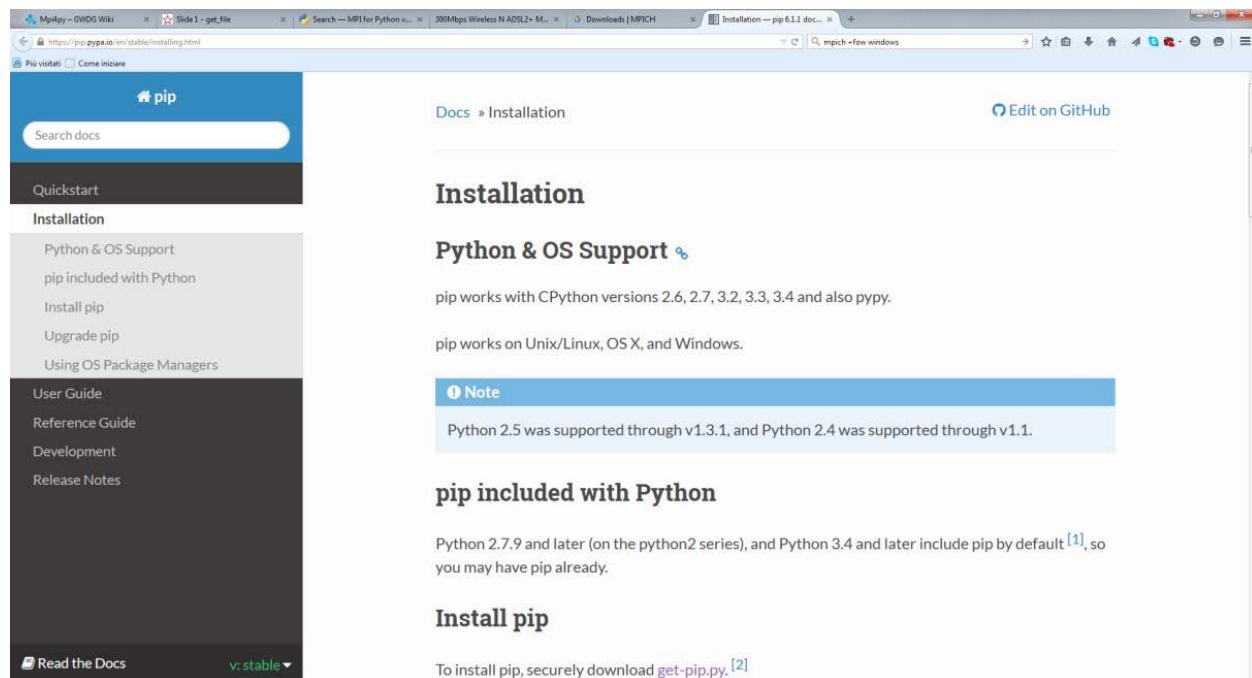
Search

**Downloads**

**MPICH is distributed under a [BSD-like license](#). NOTE: MPICH binary packages are available in many UNIX distributions and for Windows.** For example, you can search for it using "yum" (on Fedora), "apt" (Debian/Ubuntu), "pkg\_add" (FreeBSD) or "port"/"brew" (Mac OS). If available for your platform, this is likely the easiest installation method since it automatically checks for dependency packages and installs them. Otherwise you can use the [installation guide](#) for installing MPICH from the source code below.

Release	Platform	Download	Size
mpich-3.1.4 (stable release)	MPICH	[ <a href="#">http</a> ]	11 MB
hydra-3.1.4 (stable release)	Hydra (mpiexec)	[ <a href="#">http</a> ]	3 MB

2. 右键图标，选择“以管理员身份运行”。
3. 以管理员身份运行 `msiexec /i mpich_installation_file.msi` 安装MPICH2。
4. 安装的时候，选择“为所有用户安装”。
5. 运行 `wmpiconfig` 存储用户名密码，使用你的windows登录的用户名和密码。
6. 添加 `C:\Program Files\MPICH2\bin` 到系统路径，无需重启。
7. 使用 `smpd- status` 检查 `smpd`，应该返回 `smpd running on $hostname$`。
8. 到 `$MPICHROOT\examples` 文件夹使用 `mpiexec -n 4 cpi` 运行 `cpi.exe` 检查运行环境。
9. 从 <https://pip.pypa.io/en/stable/installing.html> 下载Python包管理工具 `pip`。它会在你的Python环境生成一个 `pip.exe`。



10. 在命令行安装 mpi4py :

C:> pip install mpi4py

### 3.11.2 如何做...

让我们通过打印“Hello world”来开始MPI之旅:

```
# hello.py
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
print("hello world from process ", rank)
```

通过下面的命令执行代码:

```
C:> mpiexec -n 5 python helloWorld_MPI.py
```

执行结果将得到如下的输出:

```
('hello world from process ', 1)
('hello world from process ', 0)
('hello world from process ', 2)
('hello world from process ', 3)
('hello world from process ', 4)
```

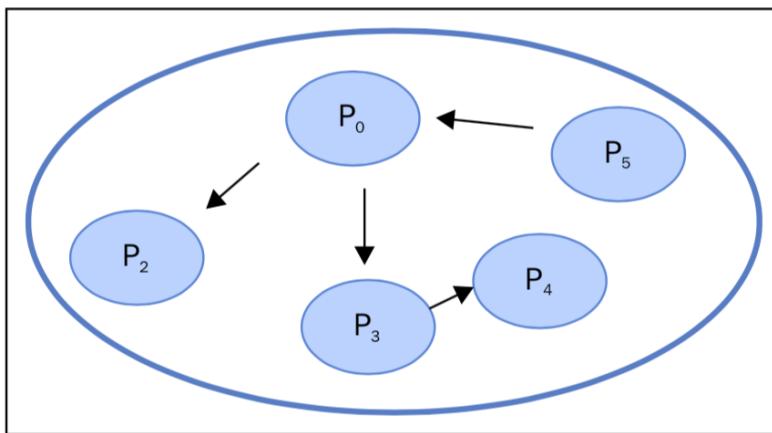
### 3.11.3 讨论

在MPI中，并行程序中不同进程用一个非负的整数来区别，叫做rank。如果我们有p个进程，那么rank会从0到p-1分配。MPI中拿到rank的函数如下：

```
rank = comm.Get_rank()
```

这个函数返回调用它的进程的rank。comm 叫做交流者（Communicator），用于区别不同的进程集合：

```
comm = MPI.COMM_WORLD
```



An example of communication between processes in MPI.COMM\_WORLD

### 3.11.4 了解更多

需要注意是，插图中的输出顺序并不是确定的，你不一定能得到插图的输出结果。多进程在同一时间启动，操作系统会决定运行的顺序。但是从中我们可以看出，MPI在每个进程中运行相同的二进制代码，每一个进程都执行相同的指令。

## 3.12 点对点通讯

MPI提供的最实用的一个特性是点对点通讯。两个不同的进程之间可以通过点对点通讯交换数据：一个进程是接收者，一个进程是发送者。

Python的mpi4py通过下面两个函数提供了点对点通讯功能：

- Comm.Send(data, process\_destination): 通过它在交流组中的排名来区分发送给不同进程的数据
- Comm.Recv(process\_source): 接收来自源进程的数据，也是通过在交流组中的排名来区分的  
Comm 变量表示交流者，定义了可以互相通讯的进程组：

```
comm = MPI.COMM_WORLD
```

### 3.12.1 如何做...

下面的例子展示了如何使用comm.send 和 comm.recv 指令在不同的进程之间交换信息。

```

from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.rank
print("my rank is : " , rank)

if rank == 0:
    data = 10000000
    destination_process = 4
    comm.send(data, dest=destination_process)
    print("sending data % s " % data + "to process % d" % destination_process)

if rank == 1:
    destination_process = 8
    data = "hello"
    comm.send(data, dest=destination_process)
    print("sending data % s :" % data + "to process % d" % destination_process)

if rank == 4:
    data = comm.recv(source = 0)
    print("data received is = % s" % data)

if rank == 8:
    data1 = comm.recv(source = 1)
    print("data1 received is = % s" % data1)

```

运行脚本的命令如下:

```
$ mpixec -n 9 python pointToPointCommunication.py
```

得到的输出如下:

```

('my rank is : ', 5)
('my rank is : ', 1)
sending data hello :to process 8
('my rank is : ', 3)
('my rank is : ', 0)
sending data 10000000 to process 4
('my rank is : ', 2)
('my rank is : ', 7)
('my rank is : ', 4)
data received is = 10000000
('my rank is : ', 8)
data1 received is = hello
('my rank is : ', 6)

```

### 3.12.2 讨论

我们将最大进程数设置为9来运行程序。所以在交流者组 `comm` 中，我们可以有9个互相通讯的进程。：

```
comm = MPI.COMM_WORLD
```

同时，我们使用 `rand` 值来区分每个进程:

```
rank = comm.rank
```

我们有两个发送者进程和两个接受者进程。`rank`值为0的进程会发送数据给`rank`值为4的接受者:

```
if rank==0:
    data= 10000000
    destination_process = 4
    comm.send(data,dest=destination_process)
```

同样的，我们可以必须指定rank值为4的进程为接收者。然后我们指定rank变量来调用 `comm.recv` 命令。

```
...
if rank == 4:
    data = comm.recv(source=0)
```

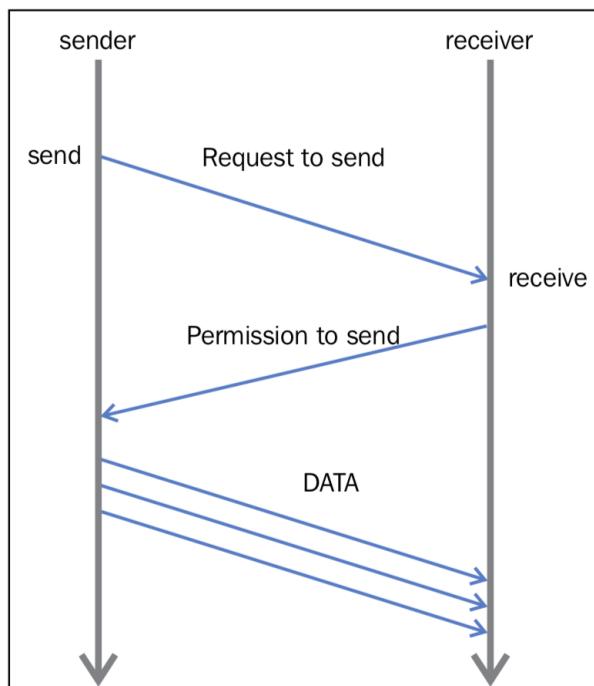
对于另外一组发送者和接收者，我们指定rank为1的作为发送者，rank为8的作为接收者。与上一组只有一点不同，这一组发送的数据类型是String。

```
if rank==1:
    destination_process = 8
    data= "hello"
    comm.send(data,dest=destination_process)
```

对于接收者来说，需要指定发送者的rank。

```
if rank==8:
    data1=comm.recv(source=1)
```

下图展示了 mpi4py 的点对点通讯协议：



The send/receive transmission protocol

整个过程分为两部分，发送者发送数据，接收者接收数据，二者必须都指定发送方/接收方。

### 3.12.3 了解更多

`comm.send()` 和 `comm.recv()` 函数都是阻塞的函数。他们会一直阻塞调用者，知道数据使用完成。同时在MPI中，有两种方式发送和接收数据：

- buffer模式
- 同步模式

在buffer模式中，只要需要发送的数据被拷贝到buffer中，执行权就会交回到主程序，此时数据并非已经发送/接收完成。在同步模式中，只有函数真正的结束发送/接收任务之后才会返回。

## 3.13 避免死锁问题

我们经常需要面临的一个问题是死锁，这是两个或多个进程都在阻塞并等待对方释放自己想要的资源的情况。`mpi4py` 没有提供特定的功能来解决这种情况，但是提供了一些程序员必须遵守的规则，来避免死锁问题。

### 3.13.1 如何做...

让我们先分析下面的Python代码，下面的代码中介绍了一种典型的死锁问题；我们有两个进程，一个 rank 等于1，另一个 rank 等于5，他们互相通讯，并且每一个都有发送和接收的函数。

```
from mpi4py import MPI
comm=MPI.COMM_WORLD
rank = comm.rank
print("my rank is : " , rank)

if rank==1:
    data_send= "a"
    destination_process = 5
    source_process = 5
    data_received=comm.recv(source=source_process)
    comm.send(data_send,dest=destination_process)
    print("sending data %s " %data_send + "to process %d" %destination_process)
    print("data received is = %s" %data_received)

if rank==5:
    data_send= "b"
    destination_process = 1
    source_process = 1
    comm.send(data_send,dest=destination_process)
    data_received=comm.recv(source=source_process)
    print("sending data %s :" %data_send + "to process %d" %destination_process)
    print("data received is = %s" %data_received)
```

### 3.13.2 讨论

如果我们尝试运行这个程序（只有两个进程足够了），会发现这两个进程都不会完成：

```
$ mpixec -n 9 python deadLockProblems.py
('my rank is : ', 8)
('my rank is : ', 3)
```

```
('my rank is : ', 2)
('my rank is : ', 7)
('my rank is : ', 0)
('my rank is : ', 4)
('my rank is : ', 6)
```

此时连个进程都在等待对方，都被阻塞住了。会发生这种情况是因为MPI的 `comm.recv()` 函数和 `comm.send()` 函数都是阻塞的。它们的调用者都在等待它们完成。对 `comm.send()` MPI来说，只有数据发出之后函数才会结束，对于 `comm.recv()` 函数来说，只有接收到数据函数才会结束。为了解决这个问题，我们可以将这连个函数这样写：

```
if rank==1:
    data_send= "a"
    destination_process = 5
    source_process = 5
    comm.send(data_send, dest=destination_process)
    data_received=comm.recv(source=source_process)
if rank==5:
    data_send= "b"
    destination_process = 1
    source_process = 1
    data_received=comm.recv(source=source_process)
    comm.send(data_send, dest=destination_process)
```

虽然这个解决方法从逻辑上纠正了，但是并不保证一定可以避免死锁问题。鉴于通讯是发生在buffer的，`comm.send()` 函数将要发送的数据完全拷贝到buffer里，只有buffer里有完整的数据之后程序才能继续运行。否则，依然会产生死锁：发送者不能发送，因为buffer已经提交但是接收到不能接收者不能接收数据，因为它被 `comm.send()` 阻塞住了。因此，我们可以交换一下发送者和接收者的顺序来解决这个问题。

```
if rank==1:
    data_send= "a"
    destination_process = 5
    source_process = 5
    comm.send(data_send, dest=destination_process)
    data_received=comm.recv(source=source_process)
if rank==5:
    data_send= "b"
    destination_process = 1
    source_process = 1
    comm.send(data_send, dest=destination_process)
    data_received=comm.recv(source=source_process)
```

最后，我们得到正确的输出如下：

```
$ mpiexec -n 9 python deadLockProblems.py
('my rank is : ', 7)
('my rank is : ', 0)
('my rank is : ', 8)
('my rank is : ', 1)
sending data a to process 5
data received is = b
('my rank is : ', 5)
sending data b :to process 1
data received is = a
('my rank is : ', 2)
('my rank is : ', 3)
('my rank is : ', 4)
```

```
('my rank is : ', 6)
```

### 3.13.3 了解更多

此并非是解决死锁问题的唯一方案。举个例子，有一个特定的函数统一了向一特定进程发消息和从一特定进程接收消息的功能，叫做 `Sendrecv`

```
Sendrecv(self, sendbuf, int dest=0, int sendtag=0, recvbuf=None, int source=0, int_  
→recvtag=0, Status status=None)
```

可以看到，这个函数的参数同 `comm.send()` MPI 以及 `comm.recv()` MPI 相同。同时在这个函数里，整个函数都是阻塞的，相比于交给子系统来负责检查发送者和接收者之间的依赖，可以避免死锁问题。用这个方案改写之前的例子如下：

```
if rank==1:  
    data_send= "a"  
    destination_process = 5  
    source_process = 5  
    data_received=comm.sendrecv(data_send,dest=destination_process,source =source_  
→process)  
if rank==5:  
    data_send= "b"  
    destination_process = 1  
    source_process = 1  
    data_received=comm.sendrecv(data_send,dest=destination_process, source=source_  
→process)
```

## 3.14 使用**broadcast**通讯

### 3.15 使用**scatter**通讯

### 3.16 使用**gather**通讯

### 3.17 使用**Alltoall**通讯

### 3.18 简化操作

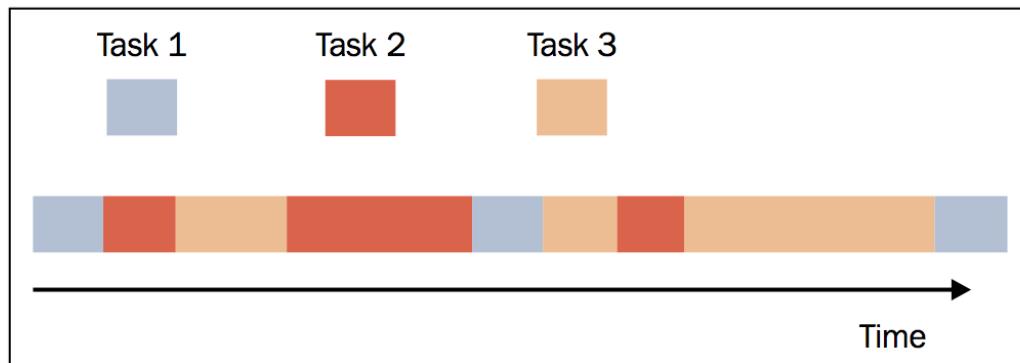
### 3.19 如何优化通讯

## 第四章 异步编程

### 4.1 介绍

除了顺序执行和并行执行的模型之外，还有第三种模型，叫做异步模型，这是事件驱动模型的基础。异步活动的执行模型可以只有一个单一的主控制流，能在单核心系统和多核心系统中运行。

在并发执行的异步模型中，许多任务被穿插在同一时间线上，所有的任务都由一个控制流执行（单一线程）。任务的执行可能被暂停或恢复，中间的这段时间线程将会去执行其他任务。下面的这幅图可以清楚地表达这个概念。



Asynchronous programming model

如上图所示，任务（不同的颜色表示不同的任务）可能被其他任务插入，但是都处在同一个线程下。这表明，当某一个任务执行的时候，其他的任务都暂停了。与多线程编程模型很大的一点不同是，多线程由操作系统决定在时间线上什么时候挂起某个活动或恢复某个活动，而在异步并发模型中，程序员必须假设线程可能在任何时间被挂起和替换。

程序员可以将任务编写成许多可以间隔执行的小步骤，这样的话如果一个任务需要另一个任务的输出，那么被依赖的任务必须接收它的输入。

## 4.2 使用Python的 `concurrent.futures` 模块

Python3.2带来了`concurrent.futures`模块，这个模块具有线程池和进程池、管理并行编程任务、处理非确定性的执行流程、进程/线程同步等功能。

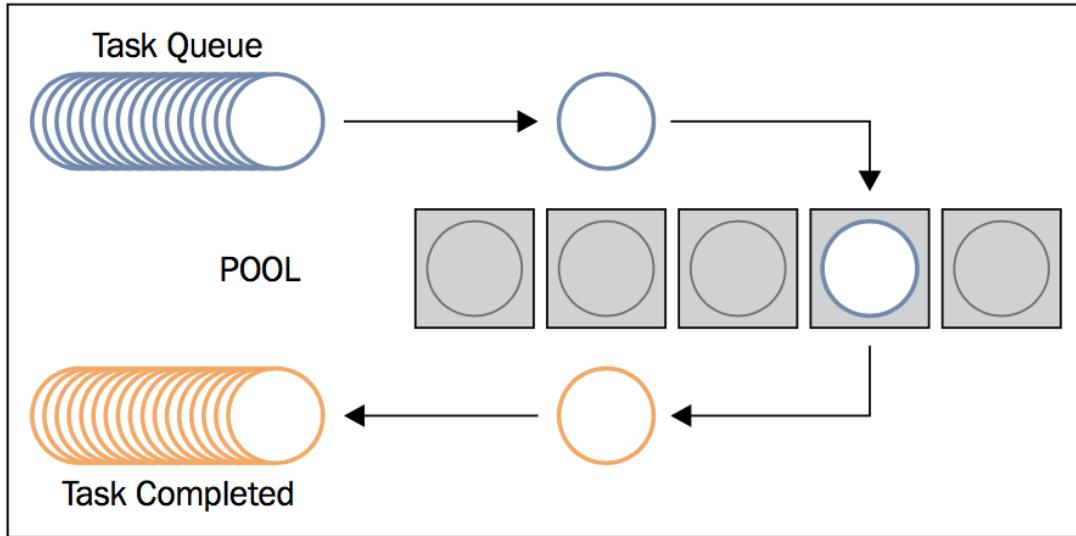
此模块由以下部分组成：

- `concurrent.futures.Executor`: 这是一个虚拟基类，提供了异步执行的方法。
- `submit(function, argument)`: 调度函数（可调用的对象）的执行，将`argument`作为参数传入。
- `map(function, argument)`: 将`argument`作为参数执行函数，以异步的方式。
- `shutdown(Wait=True)`: 发出让执行者释放所有资源的信号。
- `concurrent.futures.Future`: 其中包括函数的异步执行。Future对象是`submit`任务（即带有参数的functions）到`executor`的实例。

`Executor`是抽象类，可以通过子类访问，即线程或进程的`ExecutorPools`。因为，线程或进程的实例是依赖于资源的任务，所以最好以“池”的形式将他们组织在一起，作为可以重用的`launcher`或`executor`。

### 4.2.1 使用线程池和进程池

线程池或进程池是用于在程序中优化和简化线程/进程的使用。通过池，你可以提交任务给`executor`。池由两部分组成，一部分是内部的队列，存放着待执行的任务；另一部分是一系列的进程或线程，用于执行这些任务。池的概念主要目的是为了重用：让线程或进程在生命周期内可以多次使用。它减少了创建创建线程和进程的开销，提高了程序性能。重用不是必须的规则，但它是程序员在应用中使用池的主要原因。



## 4.2.2 准备工作

`current.Futures` 模块提供了两种 `Executor` 的子类，各自独立操作一个线程池和一个进程池。这两个子类分别是：

- `concurrent.futures.ThreadPoolExecutor(max_workers)`
- `concurrent.futures.ProcessPoolExecutor(max_workers)`

`max_workers` 参数表示最多有多少个worker并行执行任务。

## 4.2.3 如何做...

下面的示例代码展示了线程池和进程池的功能。这里的任务是，给一个list `number_list`，包含1到10。对list中的每一个数字，乘以 $1+2+3\dots+10000000$ 的和（这个任务只是为了消耗时间）。

下面的代码分别测试了：

- 顺序执行
- 通过有5个worker的线程池执行
- 通过有5个worker的进程池执行

（译者注：原文的代码是错误的，这里贴出的代码以及运行结果是修改后的，详见：关于第四章第2节书中程序的疑问 #16，感谢 @Microndgt 提出） 代码如下：：

```
import concurrent.futures
import time
number_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

def evaluate_item(x):
    # 计算总和，这里只是为了消耗时间
    result_item = count(x)
    # 打印输入和输出结果
    return result_item

def count(number) :
    for i in range(0, 10000000):
        i=i+1
    return i * number

if __name__ == "__main__":
    # 顺序执行
    start_time = time.time()
    for item in number_list:
        print(evaluate_item(item))
    print("Sequential execution in " + str(time.time() - start_time), "seconds")
    # 线程池执行
    start_time_1 = time.time()
    with concurrent.futures.ThreadPoolExecutor(max_workers=5) as executor:
        futures = [executor.submit(evaluate_item, item) for item in number_list]
        for future in concurrent.futures.as_completed(futures):
            print(future.result())
    print ("Thread pool execution in " + str(time.time() - start_time_1), "seconds")
    # 进程池
    start_time_2 = time.time()
```

```

    with concurrent.futures.ProcessPoolExecutor(max_workers=5) as executor:
        futures = [executor.submit(evaluate_item, item) for item in number_
list]
        for future in concurrent.futures.as_completed(futures):
            print(future.result())
    print ("Process pool execution in " + str(time.time() - start_time_2),
"seconds")

```

运行这个代码，我们可以看到运行时间的输出：

```

$ python3 pool.py
10000000
20000000
30000000
40000000
50000000
60000000
70000000
80000000
90000000
100000000
Sequential execution in 7.936585903167725 seconds
10000000
30000000
40000000
20000000
50000000
70000000
90000000
100000000
80000000
60000000
Thread pool execution in 7.633088827133179 seconds
40000000
50000000
10000000
30000000
20000000
70000000
90000000
60000000
80000000
100000000
Process pool execution in 4.787093639373779 seconds

```

#### 4.2.4 讨论

我们创建了一个list存放10个数字，然后使用一个循环计算从1加到10000000，打印出和与 number\_list 的乘积。：

```

def evaluate_item(x):
    # 计算总和，这里只是为了消耗时间
    result_item = count(x)
    # 打印输入和输出结果
    print ("item " + str(x) + " result " + str(result_item))

```

```
def count(number) :
    for i in range(0, 10000000):
        i=i+1
    return i * number
```

在主要程序中，我们先使用顺序执行跑了一次程序：：

```
if __name__ == "__main__":
    # 顺序执行
    start_time = time.clock()
    for item in number_list:
        evaluate_item(item)
    print("Sequential execution in " + str(time.clock() - start_time), "seconds")
```

然后，我们使用了 `futures.ThreadPoolExecutor` 模块的线程池跑了一次：：

```
with concurrent.futures.ThreadPoolExecutor(max_workers=5) as executor:
    for item in number_list:
        executor.submit(evaluate_item, item)
print ("Thread pool execution in " + str(time.clock() - start_time_1), "seconds")
```

`ThreadPoolExecutor` 使用线程池中的一个线程执行给定的任务。池中一共有5个线程，每一个线程从池中取得一个任务然后执行它。当任务执行完成，再从池中拿到另一个任务。

当所有的任务执行完成后，打印出执行用的时间：：

```
print ("Thread pool execution in " + str(time.clock() - start_time_1), "seconds")
```

最后，我们又用 `ProcessPoolExecutor` 跑了一次程序：：

```
with concurrent.futures.ProcessPoolExecutor(max_workers=5) as executor:
    for item in number_list:
        executor.submit(evaluate_item, item)
```

如同 `ThreadPoolExecutor` 一样，`ProcessPoolExecutor` 是一个executor，使用一个线程池来并行执行任务。然而，和 `ThreadPoolExecutor` 不同的是，`ProcessPoolExecutor` 使用了多核处理的模块，让我们可以不受GIL的限制，大大缩短执行时间。

## 4.2.5 了解更多

几乎所有需要处理多个客户端请求的服务应用都会使用池。然而，也有一些应用要求任务需要立即执行，或者要求对任务的线程有更多的控制权，这种情况下，池不是一个最佳选择。

## 4.3 使用Asyncio管理事件循环

Python的Asyncio模块提供了管理事件、协程、任务和线程的方法，以及编写并发代码的原语。此模块的主要组件和概念包括：

- **事件循环**: 在Asyncio模块中，每一个进程都有一个事件循环。
- **协程**: 这是子程序的泛化概念。协程可以在执行期间暂停，这样就可以等待外部的处理（例如IO）完成之后，从之前暂停的地方恢复执行。
- **Futures**: 定义了 Future 对象，和 `concurrent.futures` 模块一样，表示尚未完成的计算。

- **Tasks:** 这是Asyncio的子类，用于封装和管理并行模式下的协程。

本节中重点讨论事件，事实上，异步编程的上下文中，事件无比重要。因为事件的本质就是异步。

### 4.3.1 什么是事件循环

在计算系统中，可以产生事件的实体叫做事件源，能处理事件的实体叫做事件处理器。此外，还有一些第三方实体叫做事件循环。它的作用是管理所有的事件，在整个程序运行过程中不断循环执行，追踪事件发生的顺序将它们放到队列中，当主线程空闲的时候，调用相应的事件处理器处理事件。最后，我们可以通过下面的伪代码来理解事件循环：

```
while (1) {
    events = getEvents();
    for (e in events)
        processEvent(e);
}
```

所有的时间都在 `while` 循环中捕捉，然后经过事件处理器处理。事件处理的部分是系统唯一活跃的部分，当一个事件处理完成，流程继续处理下一个事件。

### 4.3.2 准备工作

Asyncio提供了一下方法来管理事件循环：

- `loop = get_event_loop()`: 得到当前上下文的事件循环。
- `loop.call_later(time_delay, callback, argument)`: 延后 `time_delay` 秒再执行 `callback` 方法。
- `loop.call_soon(callback, argument)`: 尽可能快调用 `callback`, `call_soon()` 函数结束，主线程回到事件循环之后就会马上调用 `callback`。
- `loop.time()`: 以float类型返回当前时间循环的内部时间。
- `asyncio.set_event_loop()`: 为当前上下文设置事件循环。
- `asyncio.new_event_loop()`: 根据此策略创建一个新的时间循环并返回。
- `loop.run_forever()`: 在调用 `stop()` 之前将一直运行。

### 4.3.3 如何做...

下面的代码中，我们将展示如何使用Asyncio库提供的时间循环创建异步模式的应用。

```
import asyncio
import datetime
import time

def function_1(end_time, loop):
    print ("function_1 called")
    if (loop.time() + 1.0) < end_time:
        loop.call_later(1, function_2, end_time, loop)
    else:
        loop.stop()

def function_2(end_time, loop):
    print ("function_2 called ")
```

```
if (loop.time() + 1.0) < end_time:
    loop.call_later(1, function_3, end_time, loop)
else:
    loop.stop()

def function_3(end_time, loop):
    print ("function_3 called")
    if (loop.time() + 1.0) < end_time:
        loop.call_later(1, function_1, end_time, loop)
    else:
        loop.stop()

def function_4(end_time, loop):
    print ("function_5 called")
    if (loop.time() + 1.0) < end_time:
        loop.call_later(1, function_4, end_time, loop)
    else:
        loop.stop()

loop = asyncio.get_event_loop()

end_loop = loop.time() + 9.0
loop.call_soon(function_1, end_loop, loop)
# loop.call_soon(function_4, end_loop, loop)
loop.run_forever()
loop.close()
```

运行结果如下：：

```
python3 event.py
function_1 called
function_2 called
function_3 called
function_1 called
function_2 called
function_3 called
function_1 called
function_2 called
function_3 called
```

#### 4.3.4 讨论

在这个例子中，我们定义了三个异步的任务，相继执行，入下图所示的顺序。



Task execution in the example

首先，我们要得到这个事件循环：

```
loop = asyncio.get_event_loop()
```

然后我们通过 `call_soon` 方法调用了 `function_1()` 函数。

```
end_time = loop.time() + 9.0
loop.call_soon(function_1, end_time, loop)
```

让我们来看一下 `function_1()` 的定义：

```
def function_1(end_time, loop):
    print ("function_1 called")
    if (loop.time() + 1.0) < end_time:
        loop.call_later(1, function_2, end_time, loop)
    else:
        loop.stop()
```

这个函数通过以下参数定义了应用的异步行为：

- `end_time`: 定义了 `function_1()` 可以运行的最长时间，并通过 `call_later` 方法传入到 `function_2()` 中作为参数
- `loop`: 之前通过 `get_event_loop()` 方法得到的事件循环

`function_1()` 的任务非常简单，只是打印出函数名字。当然，里面也可以写非常复杂的操作。

```
print ("function_1 called")
```

任务执行结束之后，它将会比较 `loop.time() + 1s` 和设定的运行时间，如果没有超过，使用 `call_later` 在 1 秒之后执行 `function_2()`。

```
if (loop.time() + 1.0) < end_time:
    loop.call_later(1, function_2, end_time, loop)
else:
    loop.stop()
```

`function_2()` 和 `function_3()` 的作用类似。

如果运行的时间超过了设定，时间循环终止。

```
loop.run_forever()
loop.close()
```

## 4.4 使用Asyncio管理协程

在上文提到的例子中，我们看到当一个程序变得很大而且复杂时，将其划分为子程序，每一部分实现特定的任务是个不错的方案。子程序不能单独执行，只能在主程序的请求下执行，主程序负责协调使用各个子程序。协程就是子程序的泛化。和子程序一样的事，协程只负责计算任务的一步；和子程序不一样的是，协程没有主程序来进行调度。这是因为协程通过管道连接在一起，没有监视函数负责顺序调用它们。在协程中，执行点可以被挂起，可以被从之前挂起的点恢复执行。通过协程池就可以插入到计算中：运行第一个任务，直到它返回(yield)执行权，然后运行下一个，这样顺着执行下去。

这种插入的控制组件就是前文介绍的事件循环。它持续追踪所有的协程并执行它们。

协程的另外一些重要特性如下：

- 协程可以有多个入口点，并可以yield多次
- 协程可以将执行权交给其他协程

yield表示协程在此暂停，并且将执行权交给其他协程。因为协程可以将值与控制权一起传递给另一个协程，所以“yield一个值”就表示将值传给下一个执行的协程。

### 4.4.1 准备工作

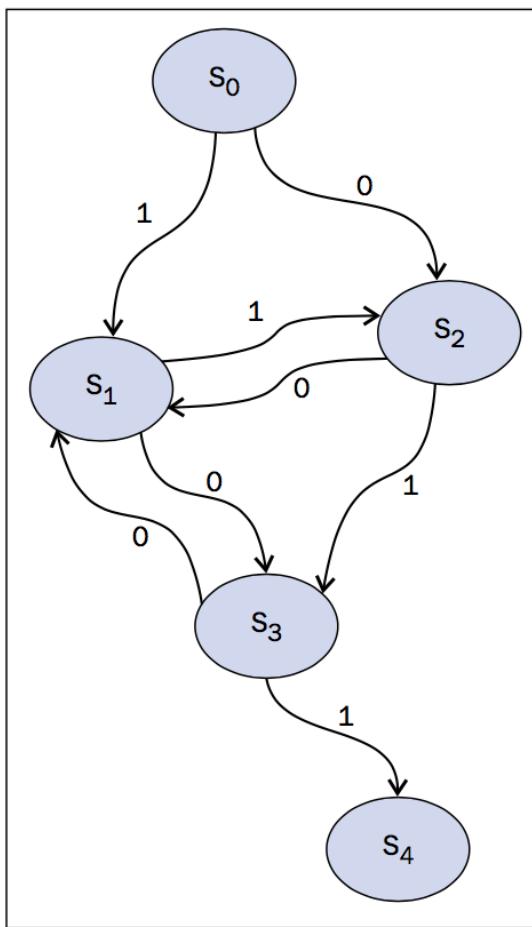
使用Asyncio定义协程非常简单，只需要一个装饰器即可：

```
import asyncio

@asyncio.coroutine
def coroutine_function(function_arguments):
    # DO_SOMETHING
```

### 4.4.2 如何做...

在这个例子中，我们将看到如何使用Asyncio的协程来模拟有限状态机。有限状态机(finite state machine or automaton, FSA)是一个数学模型，不仅在工程领域应用广泛，在科学领域也很著名，例如数学和计算机科学等。我们要模拟的状态机如下图所示：



Finite state machine

在上图中，可以看到我们的系统有 **S1, S2, S3, S4** 四个状态，**0** 和 **1** 是状态机可以从一个状态到另一个状态的值（这个过程叫做转换）。例如在本实验中，只有当只为1的时候，**S0** 可以转换到 **S1**，当只为0的时候，**S0** 可以转换到 **S2**。Python代码如下，状态模拟从 **S0** 开始，叫做 **初始状态**，最后到 **S4**，叫做 **结束状态**。

```

# Asyncio Finite State Machine
import asyncio
import time
from random import randint

@asyncio.coroutine
def StartState():
    print("Start State called \n")
    input_value = randint(0, 1)
    time.sleep(1)
    if (input_value == 0):
        result = yield from State2(input_value)
    else:
        result = yield from State1(input_value)
    print("Resume of the Transition : \nStart State calling " + result)
  
```

```

@asyncio.coroutine
def State1(transition_value):
    outputValue = str("State 1 with transition value = %s \n" % transition_value)
    input_value = randint(0, 1)
    time.sleep(1)
    print("...Evaluating...")
    if input_value == 0:
        result = yield from State3(input_value)
    else :
        result = yield from State2(input_value)
    result = "State 1 calling " + result
    return outputValue + str(result)

@asyncio.coroutine
def State2(transition_value):
    outputValue = str("State 2 with transition value = %s \n" % transition_value)
    input_value = randint(0, 1)
    time.sleep(1)
    print("...Evaluating...")
    if (input_value == 0):
        result = yield from State1(input_value)
    else :
        result = yield from State3(input_value)
    result = "State 2 calling " + result
    return outputValue + str(result)

@asyncio.coroutine
def State3(transition_value):
    outputValue = str("State 3 with transition value = %s \n" % transition_value)
    input_value = randint(0, 1)
    time.sleep(1)
    print("...Evaluating...")
    if (input_value == 0):
        result = yield from State1(input_value)
    else :
        result = yield from EndState(input_value)
    result = "State 3 calling " + result
    return outputValue + str(result)

@asyncio.coroutine
def EndState(transition_value):
    outputValue = str("End State with transition value = %s \n" % transition_value)
    print("...Stop Computation...")
    return outputValue

if __name__ == "__main__":
    print("Finite State Machine simulation with Asyncio Coroutine")
    loop = asyncio.get_event_loop()
    loop.run_until_complete(StartState())

```

运行代码，我们可以看到类似以下输出（译注，运行结果随机，这里为译者运行的三次结果）。

```

$ python3 coroutines.py
Finite State Machine simulation with Asyncio Coroutine
Start State called

...Evaluating...

```

```
....Evaluating...
....Evaluating...
....Evaluating...
....Evaluating...
....Evaluating...
....Stop Computation...
Resume of the Transition :
Start State calling State 2 with transition value = 0
State 2 calling State 1 with transition value = 0
State 1 calling State 2 with transition value = 1
State 2 calling State 1 with transition value = 0
State 1 calling State 2 with transition value = 1
State 2 calling State 3 with transition value = 1
State 3 calling End State with transition value = 1

$ python3 coroutines.py
Finite State Machine simulation with Asyncio Coroutine
Start State called

....Evaluating...
....Evaluating...
....Stop Computation...
Resume of the Transition :
Start State calling State 2 with transition value = 0
State 2 calling State 3 with transition value = 1
State 3 calling End State with transition value = 1

$ python3 coroutines.py
Finite State Machine simulation with Asyncio Coroutine
Start State called

....Evaluating...
....Evaluating...
....Evaluating...
....Evaluating...
....Evaluating...
....Evaluating...
....Evaluating...
....Stop Computation...
Resume of the Transition :
Start State calling State 1 with transition value = 1
State 1 calling State 2 with transition value = 1
State 2 calling State 1 with transition value = 0
State 1 calling State 3 with transition value = 0
State 3 calling State 1 with transition value = 0
State 1 calling State 2 with transition value = 1
State 2 calling State 3 with transition value = 1
State 3 calling End State with transition value = 1
```

### 4.4.3 讨论

每一个状态都由装饰器装饰:

```
@asyncio.coroutine
```

例如， **S0** 的定义如下所示:

```
@asyncio.coroutine
def StartState():
    print("Start State called \n")
    input_value = randint(0, 1)
    time.sleep(1)
    if (input_value == 0):
        result = yield from State2(input_value)
    else:
        result = yield from State1(input_value)
    print("Resume of the Transition : \nStart State calling " + result)
```

通过 random 模块的 randint(0, 1) 函数生成了 input\_value 的值，决定了下一个转换状态。此函数随机生成1或0：

```
input_value = randint(0, 1)
```

得到 input\_value 的值之后，通过 yield from 命令调用下一个协程。

```
if (input_value == 0):
    result = yield from State2(input_value)
else:
    result = yield from State1(input_value)
```

result 是下一个协程返回的string，这样我们在计算的最后就可以重新构造出计算过程。

启动事件循环的代码如下：

```
if __name__ == "__main__":
    print("Finite State Machine simulation with Asyncio Coroutine")
    loop = asyncio.get_event_loop()
    loop.run_until_complete(StartState())
```

## 4.5 使用Asyncio控制任务

Asyncio是用来处理事件循环中的异步进程和并发任务执行的。它还提供了 `asyncio.Task()` 类，可以在任务中使用协程。它的作用是，在同一事件循环中，运行某一个任务的同时可以并发地运行多个任务。当协程被包在任务中，它会自动将任务和事件循环连接起来，当事件循环启动的时候，任务自动运行。这样就提供了一个可以自动驱动协程的机制。

### 4.5.1 准备工作

Asyncio模块为我们提供了 `asyncio.Task(coroutine)` 方法来处理计算任务，它可以调度协程的执行。任务对协程对象在事件循环的执行负责。如果被包裹的协程要从future yield，那么任务会被挂起，等待future的计算结果。

当future计算完成，被包裹的协程将会拿到future返回的结果或异常（exception）继续执行。另外，需要注意的是，事件循环一次只能运行一个任务，除非还有其它事件循环在不同的线程并行运行，此任务才有可能和其他任务并行。当一个任务在等待future执行的期间，事件循环会运行一个新的任务。

```
"""
Asyncio using Asyncio.Task to execute three math function in parallel
"""

import asyncio
@asyncio.coroutine
```

```

def factorial(number):
    f = 1
    for i in range(2, number + 1):
        print("Asyncio.Task: Compute factorial(%s)" % (i))
        yield from asyncio.sleep(1)
        f *= i
    print("Asyncio.Task - factorial(%s) = %s" % (number, f))

@asyncio.coroutine
def fibonacci(number):
    a, b = 0, 1
    for i in range(number):
        print("Asyncio.Task: Compute fibonacci (%s)" % (i))
        yield from asyncio.sleep(1)
        a, b = b, a + b
    print("Asyncio.Task - fibonacci(%s) = %s" % (number, a))

@asyncio.coroutine
def binomialCoeff(n, k):
    result = 1
    for i in range(1, k+1):
        result = result * (n-i+1) / i
        print("Asyncio.Task: Compute binomialCoeff (%s)" % (i))
        yield from asyncio.sleep(1)
    print("Asyncio.Task - binomialCoeff(%s , %s) = %s" % (n, k, result))

if __name__ == "__main__":
    tasks = [asyncio.Task(factorial(10)),
             asyncio.Task(fibonacci(10)),
             asyncio.Task(binomialCoeff(20, 10))]
    loop = asyncio.get_event_loop()
    loop.run_until_complete(asyncio.wait(tasks))
    loop.close()

```

## 4.5.2 如何做...

在下面的代码中，我们展示了三个可以被 `Asyncio.Task()` 并发执行的数学函数。

运行的结果如下：

```

python3 task.py
Asyncio.Task: Compute factorial(2)
Asyncio.Task: Compute fibonacci (0)
Asyncio.Task: Compute binomialCoeff (1)
Asyncio.Task: Compute factorial(3)
Asyncio.Task: Compute fibonacci (1)
Asyncio.Task: Compute binomialCoeff (2)
Asyncio.Task: Compute factorial(4)
Asyncio.Task: Compute fibonacci (2)
Asyncio.Task: Compute binomialCoeff (3)
Asyncio.Task: Compute factorial(5)
Asyncio.Task: Compute fibonacci (3)
Asyncio.Task: Compute binomialCoeff (4)
Asyncio.Task: Compute factorial(6)
Asyncio.Task: Compute fibonacci (4)
Asyncio.Task: Compute binomialCoeff (5)
Asyncio.Task: Compute factorial(7)

```

```

Asyncio.Task: Compute fibonacci (5)
Asyncio.Task: Compute binomialCoeff (6)
Asyncio.Task: Compute factorial(8)
Asyncio.Task: Compute fibonacci (6)
Asyncio.Task: Compute binomialCoeff (7)
Asyncio.Task: Compute factorial(9)
Asyncio.Task: Compute fibonacci (7)
Asyncio.Task: Compute binomialCoeff (8)
Asyncio.Task: Compute factorial(10)
Asyncio.Task: Compute fibonacci (8)
Asyncio.Task: Compute binomialCoeff (9)
Asyncio.Task - factorial(10) = 3628800
Asyncio.Task: Compute fibonacci (9)
Asyncio.Task: Compute binomialCoeff (10)
Asyncio.Task - fibonacci(10) = 55
Asyncio.Task - binomialCoeff(20 , 10) = 184756.0

```

### 4.5.3 讨论

在这个例子中，我们定义了三个协程，`factorial`, `fibonacci` 和 `binomialCoeff`，每一个都带有`asyncio.coroutine`装饰器：

```

@asyncio.coroutine
def factorial(number):
    do Something

@asyncio.coroutine
def fibonacci(number):
    do Something

@asyncio.coroutine
def binomialCoeff(n, k):
    do Something

```

为了能并行执行这三个任务，我们将其放到一个task的list中：

```

if __name__ == "__main__":
    tasks = [asyncio.Task(factorial(10)),
             asyncio.Task(fibonacci(10)),
             asyncio.Task(binomialCoeff(20, 10))]

```

得到事件循环：

```
loop = asyncio.get_event_loop()
```

然后运行任务：

```
loop.run_until_complete(asyncio.wait(tasks))
```

这里，`asyncio.wait(tasks)` 表示运行直到所有给定的协程都完成。

最后，关闭事件循环：

```
loop.close()
```

## 4.6 使用Asyncio和Futures

Asyncio 模块的另一个重要的组件是 Future 类。它和 concurrent.futures.Futures 很像，但是针对Asyncio的事件循环做了很多定制。asyncio.Futures 类代表还未完成的结果（有可能是一个Exception）。所以综合来说，它是一种抽象，代表还没有做完的事情。

实际上，必须处理一些结果的回调函数被加入到了这个类的实例中。

### 4.6.1 准备工作

要操作Asyncio中的 Future，必须进行以下声明：

```
import asyncio
future = asyncio.Future()
```

基本的方法有：

- cancel(): 取消future的执行，调度回调函数
- result(): 返回future代表的结果
- exception(): 返回future中的Exception
- add\_done\_callback(fn): 添加一个回调函数，当future执行的时候会调用这个回调函数
- remove\_done\_callback(fn): 从“call when done”列表中移除所有callback的实例
- set\_result(result): 将future标为执行完成，并且设置result的值
- set\_exception(exception): 将future标为执行完成，并设置Exception

### 4.6.2 如何做...

下面的例子展示了 Future 类是如何管理两个协程的，第一个协程 first\_coroutine 计算前n个数的和，第二个协程 second\_coroutine 计算n的阶乘，代码如下：

```
# -*- coding: utf-8 -*-

"""
Asyncio.Futures - Chapter 4 Asynchronous Programming
"""

import asyncio
import sys

@asyncio.coroutine
def first_coroutine(future, N):
    """前n个数的和"""
    count = 0
    for i in range(1, N + 1):
        count = count + i
        yield from asyncio.sleep(4)
    future.set_result("first coroutine (sum of N integers) result = " + str(count))

@asyncio.coroutine
def second_coroutine(future, N):
    count = 1
    for i in range(2, N + 1):
        count *= i
        yield from asyncio.sleep(4)
    future.set_result("second coroutine (product of N integers) result = " + str(count))
```

```

        count *= i
    yield from asyncio.sleep(3)
    future.set_result("second coroutine (factorial) result = " + str(count))

def got_result(future):
    print(future.result())

if __name__ == "__main__":
    N1 = int(sys.argv[1])
    N2 = int(sys.argv[2])
    loop = asyncio.get_event_loop()
    future1 = asyncio.Future()
    future2 = asyncio.Future()
    tasks = [
        first_coroutine(future1, N1),
        second_coroutine(future2, N2)]
    future1.add_done_callback(got_result)
    future2.add_done_callback(got_result)
    loop.run_until_complete(asyncio.wait(tasks))
    loop.close()

```

输出如下:

```

$ python asy.py 1 1
first coroutine (sum of N integers) result = 1
second coroutine (factorial) result = 1
$ python asy.py 2 2
first coroutine (sum of N integers) result = 3
second coroutine (factorial) result = 2
$ python asy.py 3 3
first coroutine (sum of N integers) result = 6
second coroutine (factorial) result = 6
$ python asy.py 4 4
first coroutine (sum of N integers) result = 10
second coroutine (factorial) result = 24

```

### 4.6.3 讨论

在主程序中，我们通过定义future对象和协程联系在一起:

```

if __name__ == "__main__":
    ...
    future1 = asyncio.Future()
    future2 = asyncio.Future()

```

定义tasks的时候，将future对象作为变量传入协程中:

```

tasks = [
    first_coroutine(future1, N1),
    second_coroutine(future2, N2)]

```

最后，添加一个future执行时的回调函数:

```

def got_result(future):
    print(future.result())

```

在我们传入future的协程中，在计算之后我们分别添加了3s、4s的睡眠时间：

```
yield from asyncio.sleep(4)
```

然后，我们将future标为完成，通过 `future.set_result()` 设置结果。

#### 4.6.4 了解更多

交换两个协程睡眠的时间，协程2会比1更早得到结果：

```
$ python asy.py 3 3
second coroutine (factorial) result = 6
first coroutine (sum of N integers) result = 6
$ python asy.py 4 4
second coroutine (factorial) result = 24
first coroutine (sum of N integers) result = 10
```



# CHAPTER 5

---

## 第五章 分布式Python编程

---

### 5.1 介绍

### 5.2 使用Celery实现分布式任务

### 5.3 如何使用Celery创建任务

### 5.4 使用SCOOP进行科学计算

### 5.5 使用SCOOP处理map函数

### 5.6 使用Pyro4进行远程方法调用

### 5.7 使用Pyro4清理对象

### 5.8 使用Pyro4部署客户端-服务器应用

### 5.9 使用PyCSP交流顺序的进程

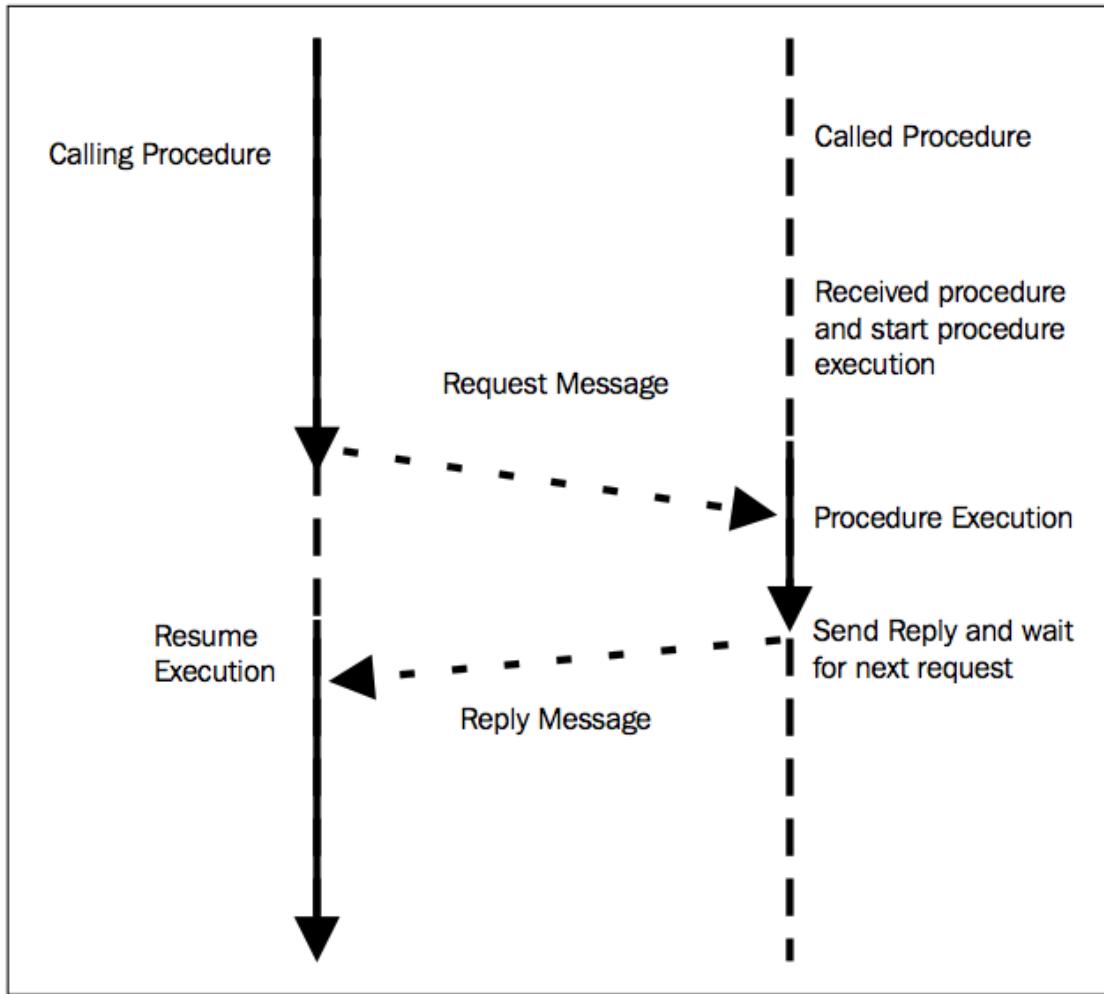
### 5.10 使用Disco进行MapReduce

### 5.11 使用RPyC远程调用

Remote Python Call(RPyC)是一个用作远程过程调用，同时也可以用作分布式计算的Python模块。其基础RPC主要是提供一种将控制从当前程序(客户端)转移到其他程序(服务器)的机制，类似于在一个主程序里

去调用一个子程序。这种方式的优点是它拥有非常简单的语义，知识以及熟悉的集中式函数调用。在一个程序调用里，客户端进程会挂起，直到服务器完成计算返回结果后才会继续执行。这种方法之所以高效是因为客户端-服务器的通信表现为过程调用而不是传输层调用，因此所有网络操作的细节通过将应用程序置于被称作存根(stubs)的本地过程对应用程序隐藏。RPyC的主要特性是：

- 语法透明，远程过程调用和本地调用有一样的语法
- 语义透明，远程过程调用和本地调用是语义一致的
- 可以处理同步和异步通信
- 对称通信协议意味着不论是客户端还是服务器都可以处理一个请求



The remote procedure call model

### 5.11.1 准备工作

使用pip来安装非常容易。在你的命令行终端里，键入下面的命令： `pip install rpyc`。

另外你可以去 <https://github.com/tomerfiliba/rpyc> 下载完整的包(是一个 .zip 文件)。下载完成之后在包的根目录里执行以下命令： `python setup.py install`。

安装完成之后，你可以浏览这个库。在我们的例子里，我们将在同一台机器 `localhost` 上运行一个客户端和服务器。使用 `rpyc` 运行一个服务器非常简单：在 `rpyc` 包的目录 `..../rpyc-master/bin` 里执行 `rpyc_classic.py`: `python rpyc_classic.py`。

在运行这个脚本之后，你可以看到在命令提示符上有如下信息：

```
INFO:SLAVE/18812:server started on [0.0.0.0]:18812
```

## 5.11.2 如何做...

我们现在已经可以开始我们第一个例子了：重定向一个远程处理的 `stdout`:

```
import rpyc
import sys
c = rpyc.classic.connect("localhost")
c.execute("print('hi python cookbook')")
c.modules.sys.stdout = sys.stdout
c.execute("print('hi here')")
```

通过运行这个脚本，你会在服务器端看到重定向的输出：

```
INFO:SLAVE/18812:server started on [0.0.0.0]:18812
INFO:SLAVE/18812:accepted 127.0.0.1:6279
INFO:SLAVE/18812:welcome [127.0.0.1]:6279
hi python cookbook
```

(译者注：在执行 `c.modules.sys.stdout = sys.stdout` 之后，`print` 将会输出到客户端的命令行里)

## 5.11.3 讨论

第一步是执行一个客户端去连接服务器：

```
import rpyc
c = rpyc.classic.connect("localhost")
```

这里，客户端的语句 `rpyc.classic.connect(host, port)` 根据给定的 `host` 和 `port` 来创建一个套接字。套接字定义了连接的端点。`rpyc` 使用套接字和其他程序通信，其可以分布在不同的计算机上。

下来，我们执行了这条语句：`c.execute("print('hi python cookbook')")`。

这条语句就会在服务器上执行 `print` 语句(远程的“exec”语句)。



# CHAPTER 6

---

## 第六章 Python GPU编程

---

[6.1 介绍](#)

[6.2 使用PyCUDA模块](#)

[6.3 如何创建一个PyCUDA应用](#)

[6.4 理解PyCuDA内存模型](#)

[6.5 使用GPUArray进行内核调用](#)

[6.6 使用PyCUDA评估元素](#)

[6.7 使用PyCUDA进行MapReduce操作](#)

[6.8 使用NumbaPro进行GPU编程](#)

[6.9 使用GPU加速的库](#)

[6.10 使用PyOpenCL模块](#)

[6.11 如何创建一个PyOpenCL应用](#)

[6.12 使用PyOpenCL评估元素](#)

[6.13 使用PyOpenCL测试你的GPU应用](#)

---

Chapter 6. 第六章 Python GPU编程

# CHAPTER 7

---

## Indices and tables

---

- search