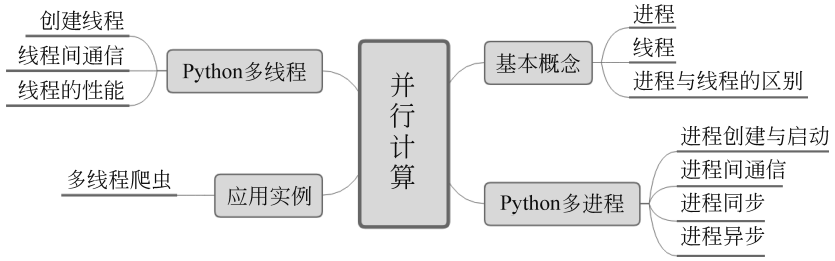
PYTHON

第 3 章

并行计算

3.1 导学



学习目标：

- 理解进程与线程的概念。
- 理解进程同步、异步和通信机制。
- 掌握 Python 进程创建与同步的方法。
- 掌握 Python 线程创建与应用的方法。

在大数据分析任务中，并行地完成数据处理和分析是一个基础要求，也是现代计算机计算性能的必要保证。虽然并行计算有复杂的理论和机制，但是通过 Python 封装的并行计算模块，可以快速实现并行计算任务。

本章在简要介绍并行计算相关理论和机制的基础上，利用典型示例，展示 Python 多进程、多线程计算的实现，并将进程的同步、异步和通信机制融合到示例中，避免枯燥讲解理论知识，降低学习和编程实现的难度。

3.2 基本概念

3.2.1 进程

程序不是进程。程序是一个静态的概念，是完成某个功能的指令集合，通常以文件的形式存储在硬盘等外部存储器上。程序不能反映系统不断变化的状态。

当运行程序以执行计算任务时，程序加载到内存中，相关代码被 CPU 执行，此时运行中的程序就是一个进程(Process)，即进程是应用程序的执行实例。现代的操作系统几乎都支持多进程并发执行。注意，并发和并行是两个概念，并行指在同一时刻有多条指令在多个处理器上同时执行；并发指在同一时刻只能有一条指令执行，但多个进程指令被快速轮换执行，使得在宏观上具有多个进程同时执行的效果。

站在操作系统资源管理的角度来看，进程是资源分配的基本单位，也是调度运行的基本单位。程序执行时，以进程为单位向操作系统申请资源，操作系统把各种资源，例如足够的内存，分配给进程。操作系统以进程为单位进行 CPU 管理和调度，例如仅有一颗 CPU 时，多个进程排队等待 CPU 的分配，操作系统基于轮转时间片方法，各进程轮流在 CPU 上运行一个指定的很短时间，然后撤下来重新排队，换其他进程执行。

在这个调度过程中，进程有 3 个基本状态：就绪、阻塞和运行，并在这些状态中转换，如图 3-1 所示。进程被“新建”时，向操作系统申请内存等必要资源，“就绪”后进入进程队列，等待分配 CPU 运行时间；获得许可后在 CPU 上“运行”；如果某个中断信号被触发，例如 I/O 请求或时间片用完了，进程被“阻塞”；阻塞条件解除后，进程重新进入“就绪”队列，继续等待 CPU。

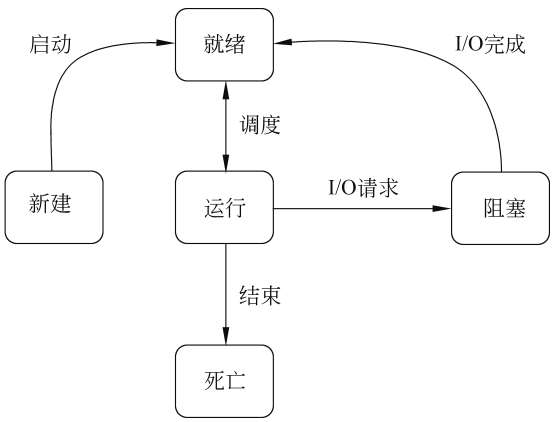


图 3-1　进程的状态变化

当进程完成计算后，释放占用的所有资源，从系统中撤销，即进入死亡状态。进程从创建到撤销的时间段就是进程的生命期。

进程是一个动态的概念，而程序是一个静态概念。不同的进程可以执行同一个程序。

3.2.2　线程

线程是进程的组成部分，一个进程可以拥有多个线程。在多线程中，会有一个主线程来完成整个进程从开始到结束的全部操作，而其他的线程会在主线程的运行过程中被创建或退出。线程是进程中执行运算的最小单位，亦即执行处理机调度的基本单位。

对于 Windows 操作系统，当进程被初始化后，主线程就被创建了，对于绝大多数的应用程序来说，通常仅要求有一个主线程，但也可以在进程内创建多个顺序执行流，这些顺序执行流就是线程。如果一个进程中只有一个线程，则叫作单线程。超过一个线程就叫作多线程。

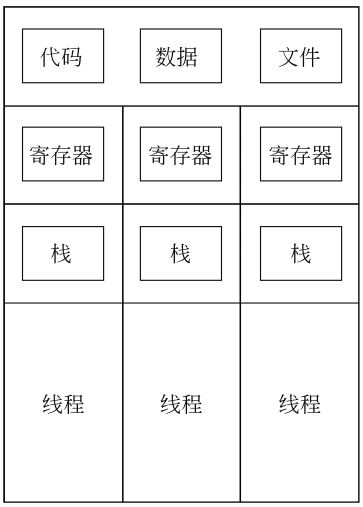


图 3-2　含多线程的进程

每个线程必须有自己的父进程，且它可以拥有自己的堆栈、程序计数器和局部变量，但不拥有系统资源，因为它和父进程的其他线程共享该进程所拥有的全部资源，如图 3-2 所示。线程可以完成一定的任务，可以与其他线程共享父进程中的共享变量及部分环境，相互协同完成进程所要完成的任务。

多个线程共享父进程中的全部资源会使得编程更加方便，需要注意的是，要确保线程不会妨碍同一进程中的其他线程。

线程是独立运行的，它并不知道进程中是否还有其他线程存在。线程的运行是抢占式的，也就是说，当前运行的线程在任何时候都可能被挂起，以便另外一个线程可以运行。

多线程也是并发执行的，即同一时刻，Python 主程序只允许有一个线程执行。

从逻辑的角度来看，多线程存在于一个应用程序中，让一个应用程序可以有多个执行部分同时执行，但操作系统无须将多个线程看作多个独立的应用，对多线程实现调度和管理以及资源分配可由进程本身负责完成。

3.2.3　进程与线程的区别

简而言之，进程和线程的关系是这样的：操作系统可以同时执行多个任务，每一个任务就是一个进程，进程可以同时执行多个任务，每一个任务就是一个线程。线程是依托父进程存在的。

一个线程可以创建和撤销另一个线程，同一进程中的多个线程之间可以并发执行。就并发执行来看，进程与线程比较相似。但是线程是个细粒度的概念，更有利于在把庞大的计算任务分解后提高 CPU 的利用率。进程像一个大型单位的各职能部门，负责完成相关职能和任务，拥有财力、物力和职权等资源；而线程就像是这个部门中的职员，负责具体任务的实施，职员所能调动的资源仅限于本部门所拥有的资源。各职能部门并行运行，彼此之间有协作，各部门的职员在部门内也是并行工作。线程更像是一个轻量级的进程。

需要特别说明的是，开发并行程序时，传统的一些代码调试技术就失效了，特别是设置

断点和单步跟踪这两个对单进程和单线程开发非常实用的技巧，因为无法用于发现并行过程设计上的缺陷和死锁等问题。调试困难是并行程序设计的一大难点。

为了降低开发难度，减少开发错误，首先推荐尽量利用各种函数编程工具来实现并行计算，例如使用 NumPy 的数组类型实现矩阵计算，NumPy 会自动进行并行计算。其次，尽量选择并发机制简洁清晰的开发工具。Python 提供了优秀的并行开发模块，仅用少量简单代码就可以实现原本复杂的并行计算。

3.3 Python 多进程

Python 提供了一个非常优秀的多进程模块 multiprocessing，支持进程的创建、管理和完成进程间通信。

3.3.1 进程创建与启动

Python 通过 Process 类来创建进程。创建类对象时通常只需传递两个基本参数：参数 target 传递新进程要执行的函数，参数 args 以元组的形式传递要执行函数的参数。

Process 类通过方法 start()启动进程，该方法每个进程对象最多运行一次。

方法 join([timeout])用于阻塞。子进程调用 join()方法，阻塞的不是自己，而是创建子进程的主进程，即主进程需等待子进程返回后，才能继续执行后续的代码。可以用参数 timeout 设置等待时间。

一个进程可以被 join 多次。进程无法 join 自身，因为会导致死锁。

下面的代码演示了创建子进程的基本方法：

```
from multiprocessing import Process

def worker(name):
    print('你好', name)

def main():
    p=Process(target=worker, args=('Python',))
    p.start()
    p.join()

if __name__=='__main__':
    main()
```

运行结果：

```
你好 Python
```

需要注意：

(1) start()之后使用才能完成 join()功能；

(2) join()不是必需的，如果不想等待，可以不调用 join()；

(3) 给 args 参数传参时,哪怕只有一个参数,也要在元组的圆括号内写上逗号,例如('Python',),这个逗号可以避免传参失败等可能的错误。

本章中的示例应该在支持多进程的集成开发环境或者命令行状态下运行,不推荐 Web 界面的 Jupyter Notebook 环境,以确保多进程程序正常运行。

每个进程被创建后,系统都会为其分配一个进程号。

接下来实现较复杂的例子。

首先导入一些必要的包。定义一个函数 sleeper(),打印进程 ID,用休眠操作来模拟程序执行,如例 3-1 所示。

【例 3-1】　创建一个子进程

```
from multiprocessing import Process
import os
import time

def sleeper(name, seconds):
    print("Process ID#%s" %(os.getpid()))
    print( "%s will sleep for %s seconds" %(name, seconds))
    time.sleep(seconds)
    print( u"子进程%s 结束。" %(os.getpid()))

def one_proc():
    child_proc=Process(target=sleeper, args=('Python', 5))
    child_proc.start()
    print( u"子进程 start 之后!")
    print( u"主进程将 join")
    child_proc.join()
    print( u"join 之后!")
    print( u"主进程的 ID 是: %s" %(os.getpid()))
    print( u"主进程结束!")

one_proc()
```

运行结果:

```
子进程 start 之后!
主进程将 join
Process ID#17916
Python will sleep for 5 seconds
子进程 17916 结束。
join 之后!
主进程的 ID 是: 12948
主进程结束!
```

例 3-1 中只创建了一个子进程,加上主进程本身,在操作系统的进程列表中可以观察到两个 Python 进程。

运行结果中的子进程 ID 和主进程 ID 是由操作系统分配和管理的，每次运行可能是不同的进程 ID。

虽然这里把新进程称为“子进程”，但是它和主进程并没有主从依赖关系，是独立运行的两个进程。

如果不执行阻塞方法，注释代码 child_proc.join()和它前后的打印语句，再次执行上面的代码，运行结果如下：

```
子进程 start 之后!
主进程的 ID 是: 1960
主进程结束!
Process ID# 14420
bob will sleep for 5 seconds
子进程 14420 结束。
```

从运行结果来看，子进程启动后，主进程没有等待子进程，继续执行了打印语句 print(u" 主进程的 ID 是：%s" % (os.getpid()))，显示了进程 ID，然后结束了运行。子进程的存在是独立于主进程的，虽然主进程结束，子进程继续正常运行，直至任务完成，正常退出。

如果需要创建更多进程，可以用循环语句实现，如例 3-2 所示。

【例 3-2】 创建多个进程

```
import random

def worker(num):
    """process/thread worker function"""
    print('Worker:', num)
    time.sleep(random.randint(1,2))

def some_procs():
    jobs=[]
    for i in range(5):
        p=Process(target=worker, args=(i,))
        #(i,)中逗号不能少
        jobs.append(p)
        p.start()
    print( 'wait joining...')
    for j in jobs:
        j.join()
    print( u'程序结束!')
some_procs()
```

运行结果：

```
wait joining...
Worker: 2
Worker: 0
Worker: 3
```

```
Worker: 1
Worker: 4
程序结束!
```

在函数 some_procs()内,用一个 for 循环语句循环生成 5 个进程,每生成一个 Process 类对象,就调用 start()方法启动该进程,并用列表 jobs 保存创建的进程对象。然后再使用一个循环,每个进程对象都执行 join()语句阻塞主进程,要求主进程等待它运行结束后返回。

为了更加逼近真实情况,导入 random 模块,用随机方法控制 worker()的 sleep 语句休眠 1～2s。

可以看到,"wait joining…"先被打印出来了,说明主进程中这条打印语句没有等待子进程的运行。然后是各子进程运行时打印的信息,从编号来看,并非是按照进程创建顺序输出的。如果多次执行这段程序,可以看到各 worker 的打印顺序是变化的。这些现象说明,这几个进程是并发运行的。

主进程因为被阻塞,所以一直等到所有子进程运行结束后,才在最后打印输出"程序结束!"。

3.3.2　进程间通信

所谓进程间通信,就是在多个进程间交换数据,例如共同使用同一个共享变量,如果一个进程修改了该共享变量,其他进程可以看到变量被修改后的新值。

那么,这个任务能否用一个全局变量来实现呢(见例 3-3)?

【例 3-3】　全局变量与多进程

```
share_num=0   #全局变量

def worker_1():
    global share_num
    share_num+=20
    print(u"worker_1,share_num=%d" %share_num)

def worker_2():
    global share_num
    share_num+=100
    print(u"worker_2,share_num=%d" %share_num)

def test_sharedata():
    p1=Process(target=worker_1)
    p2=Process(target=worker_2)
    p1.start()
    p1.join()
    print(u"p1启动后,主进程里 share_num=%d" %share_num)
    p2.start()
    p2.join()
```

```
    print(u"p2 启动后,主进程里 share_num=%d" %share_num)
test_sharedata()
```

运行结果：

```
worker_1,share_num=20
p1 启动后,主进程里 share_num=0
worker_2,share_num=100
p2 启动后,主进程里 share_num=0
```

例 3-3 中定义了两个子进程要执行的函数,其中,worker_1()对全局变量 share_num 加 20,worker_2()对全局变量 share_num 加 100。

在 test_sharedata()函数中,首先启动 worker_1()对应的进程 p1 并阻塞,再启动 worker_2()对应的进程 p2 并阻塞。因为启动和阻塞成对出现,这里的 p2 进程须等待 p1 结束后才能运行。

注意：把 start()和 join()语句放在恰当的位置,可以控制进程的执行顺序。

从运行结果来看,worker_1 虽然修改了全局变量 share_num,但是主进程里的 share_num 仍然是 0;worker_2 执行后,给全局变量 share_num 加了 100,但是 share_num 的数值只是 100,并没有累加 worker_1 加的 20;主进程再次打印 share_num,仍然是 0。

其原因必须从进程的概念和原理上来理解。人们说“进程是资源分配的基本单位”,其中资源包括 CPU、内存、打印机等各类计算机资源。为了保证每个进程安全稳定地运行,操作系统规定,每个进程都在各自独立的内存空间运行,不能直接访问其他进程的内存。在独立的空间运行表示每个进程都需要单独申请内存空间,它的所有变量都在自己的内存空间里。注意变量存在的本质是有一段分配给它的内存,不是用变量名来区分不同变量,而是用不同的内存地址来区分。不能直接访问其他进程的内存,可以最大程度上保护一个进程不被其他进程的错误所干扰,或被其他进程破坏。这也意味着,不可以直接读取其他进程的变量数值。可以把进程的内存空间形象地看作进程的“家”,家里有一群小孩子(变量),“家”是私有领地,不允许其他进程进来把孩子领走。

那么全局变量为什么也没实现进程间数据共享呢？因为这里的全局变量指的是作用范围是整个程序内的变量,而这个程序运行时是以一个进程的形式存在的。所以全局变量仍然是“家”里的孩子,只不过可以在所有房间“串门”,但是不允许其出门。

每个进程都是独立的,即便是执行相同的函数,只不过是功能相同而已,不代表它们共享同一段内存。所以全局变量只是相对于一个进程而言的,不是多个进程之间的全局变量,因此不能起到多进程间共享数据的作用。

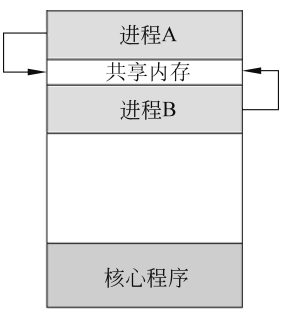


图 3-3　利用共享内存通信

怎么解决多进程间共享数据的问题呢？操作系统创建了一段可供各个进程共享的专用内存空间,如图 3-3 所示。但是多进程并发访问同一段内存时,有的要读数据,有的要改数据,有的先,有的后,需要管理协调。操作系统根据不同的访问机制管理共享内存的访问,协调这些进程的并发访问。这些机制包括信号量、锁、管程等。这种协调各进程并发工作的机制称为进程

同步。

正确灵活地应用这些进程同步机制，是成功实施并行计算的必要条件。

Python 的 multiprocessing 模块提供了共享变量和共享数组类型。但是更加实用的数据结构是支持多进程安全访问的队列、管道等。

1. 共享变量和数组

【例 3-4】　共享变量的应用

```
from multiprocessing import Process,Value,Array
def fun_memory(n, a):
    n.value=3.1415927
    for i in range(len(a)):
        a[i]=-a[i]

def share_memory():
    num=Value('d', 0.0)
    arr=Array('i', range(10))

    p=Process(target=fun_memory, args=(num, arr))
    p.start()
    p.join()      #注意,此处的阻塞是必要的

    print(f'共享变量的值={num.value}')
print(f'共享数组的值={arr[:]}')

share_memory()
```

运行结果：

```
共享变量的值=3.1415927
共享数组的值=[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

例 3-4 中声明了一个共享变量 num 和共享数组 arr，创建了一个进程，修改了 num 和 arr 的值，然后回到主进程，打印主进程的 num 和 arr，发现子进程对这两个变量的修改可以被主进程查看。这说明这两个变量实现了在两个进程间的共享。

注意：p.join()语句是不可少的。如果将这条语句写入注释，则主进程不会等待子进程的运行，直接打印 num 和 arr，那么只能看到这两个变量初始的值，无法观察到子进程对它们的修改。

2. 管道

用管道实现进程间数据共享。管道是一个线性结构，一端有一个发送者，另一端有一个接收者。对于单向管道，固定一方发送消息，另一方接收消息。而 Python 提供的是双向管道，两端都可以发送和接收。

【例 3-5】　单向管道通信

```
from multiprocessing import Process,Pipe
```

```
def proc1(pipe):
    #while True:
    for i in range(10):
        print( "proc1 发送: %s" %(i))
        pipe.send(i)
        time.sleep(1)

def proc2(pipe):
    #while True:
    for i in range(5):
        print( "proc2 接收:", pipe.recv())
        time.sleep(1)

def procs_pipe():
    parent_conn, child_conn  =Pipe()
    p1=Process(target=proc1, args=(parent_conn,))
    p2=Process(target=proc2, args=(child_conn,))

    p1.start()
    p2.start()

    p1.join()
p2.join()

procs_pipe()
```

运行结果：

```
proc1 发送: 0
proc2 接收: 0
proc1 发送: 1
proc2 接收: 1
proc1 发送: 2
proc2 接收: 2
proc1 发送: 3
proc2 接收: 3
proc1 发送: 4
proc2 接收: 4
proc1 发送: 5
proc1 发送: 6
proc1 发送: 7
proc1 发送: 8
proc1 发送: 9
```

从例 3-5 运行结果可以看到，发送和接收成对出现，一发一收。接收者 proc2 收到 5 次数据后退出，发送方继续发送了 5 次消息。

3. 队列

队列是一种线性数据结构，允许用 put()方法在队列尾部插入新数据，用 get()方法从队列首部提取一个数据元素。使用这种多进程安全的队列，需要先导入相应的类：

```
from multiprocessing import Queue
```

【例 3-6】　利用队列，实现多进程间通信

```
from multiprocessing import Process,Queue

def subproc_queue(queue):
    print('子进程 sleeping...')
    time.sleep(5)
    print('子进程醒了，开始干活。')
    queue.put([42,'20209528','郝雪生'])
    print('子进程又 sleeping...')
    time.sleep(2)

def procs_queue():
    queue=Queue()
    p=Process(target=subproc_queue, args=(queue,))
    p.start()
    print('主进程从队列取数据，若没有数据则等待...')
    val=queue.get()
    print('主进程获得数据：%s' %str(val))
    p.join()
    print('OVER!')2

procs_queue()
```

运行结果：

```
主进程从队列取数据，若没有数据则等待...
子进程 sleeping...
子进程醒了，开始干活。
子进程又 sleeping...
主进程获得数据：[42, '20209528', '郝雪生']
OVER!
```

例 3-6 的运行结果显示，通过队列主进程获得了子进程的数据。使用队列是实现进程间通信的一种稳定、简便的方式。

3.3.3　进程同步

进程同步指协调多个相关进程的执行顺序，使并发执行的各进程能按照一定的规则或者时序共享访问系统资源，得到合理的结果。

这里的共享资源往往是互斥的，这类资源被称为临界资源，可以是硬件资源，也可以是

软件资源。例如,一个进程要写文件,同时另一个进程要读同一个文件。一读一写,显然是冲突的,必须协调一个顺序。要么等读进程读完文件,换写进程来写,要么先写后读。

1. 信号量

协调访问互斥的临界资源的一个基本机制是信号量。信号量会记录自己还剩几个可用资源,每被申请一次就减 1,若小于 0 就阻塞申请资源的进程;若有进程释放资源,信号量就把可用资源数加 1,原来被阻塞的进程就可以获得资源,得以运行。

【例 3-7】 信号量

```
from multiprocessing import Process,Semaphore
def test_Semaphore(i,sem):
    #临界区开始
    sem.acquire()    #申请信号量
    print(u'%s 获得信号量' %i)
    time.sleep(random.randint(1,5))
    sem.release()
    #临界区结束
    print(u'%s 释放信号量' %i)


def Semaphore_Demo():
    sem=Semaphore(1)  #信号量可用来控制对共享资源的访问数量
    for i in range(2):
        p=Process(target=test_Semaphore,args=(i,sem))
        p.start()
Semaphore_Demo()
```

运行结果:

```
0 获得信号量
0 释放信号量
1 获得信号量
1 释放信号量
2 获得信号量
2 释放信号量
3 获得信号量
3 释放信号量
```

从例 3-7 可以看到,函数 Semaphore_Demo()里并没有执行 join()语句阻塞任何进程,但是执行结果显然是串行的,4 个进程按顺序依次执行。一个进程必须获得信号量后才能运行,直到该进程释放信号量后,其他进程才能有机会获得信号量,进而才能获得执行权限。本例中只有一个信号量,所以 4 个进程只能依次运行,当一个进程运行时,其他进程等待。

信号量的数量对应了互斥资源的数量。如果增大信号量和进程的数量,例如 3 个信号量和 6 个进程,代码修改如下:

```
def Semaphore_Demo():
```

```
    sem=Semaphore(3)  #信号量可用来控制对共享资源的访问数量
    for i in range(6):
        p=Process(target=test_Semaphore,args=(i,sem))
        p.start()
```

运行结果：

```
运行结果为:
0 获得信号量
1 获得信号量
2 获得信号量
1 释放信号量
3 获得信号量
0 释放信号量
4 获得信号量
3 释放信号量
5 获得信号量
2 释放信号量
4 释放信号量
5 释放信号量
```

需要说明的是，上述申请和释放信号量的语句是存在风险的。如果一个获得信号量的进程在释放信号量之前发生异常错误退出了，则该信号量没有被正常释放，其他进程也就无法正常获得该信号量，可能造成其他进程一直等待而无法执行，产生“死锁”。

2. 锁

下面再演示另一种基本同步机制——“锁”，用于协调进程执行顺序。与信号量类似，必须先调用方法 acquire()申请锁，得到锁后才能继续执行，否则继续等待，用完后必须调用方法 release()释放锁。

【例 3-8】　锁机制

```
from multiprocessing import Process, Lock
def print_info(lock, id):
    lock.acquire()
    try:
        print(f'{id}说: Hello Python ...')
    finally:
        lock.release()

def lock_demo():
    #加锁
    lock=Lock()
    for num in range(5):
        Process(target=print_info, args=(lock, num)).start()

lock_demo()
```

运行结果：

```
0 说: Hello Python ...
1 说: Hello Python ...
2 说: Hello Python ...
3 说: Hello Python ...
4 说: Hello Python ...
```

从例 3-8 可以看到，在锁的协调下，各进程依次打印。如果不协调打印顺序，在实际环境中，可能会出现各进程打印的信息彼此干扰、信息混乱的情况。

这个例子中，为锁的申请和释放增加了异常处理机制，并用 finally 语句确保任何异常发生时，都要在释放锁之后再结束函数的调用。

3. 同步模型

一些经典进程同步模型有利于解决同步问题，例如生产者-消费者，读者-写者，哲学家就餐等。下面以生产者-消费者模型为例讲解同步模型。

生产者-消费者模型的描述为：若干生产者进程负责生产产品，若干消费者进程负责消费这些产品，为了协调双方的动作，在双方之间设置一个可放 n 个产品的传送带（缓冲区）。缓冲区空，则消费者被阻塞，等待；缓冲区满，则生产者被阻塞，如图 3-4 所示。

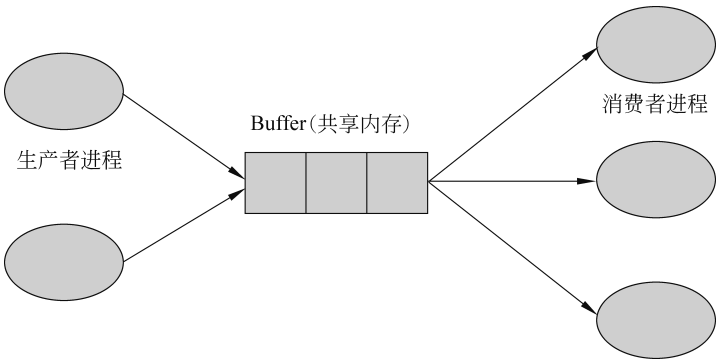


图 3-4　生产者-消费者模型

【例 3-9】　生产者-消费者模型

```
from multiprocessing import Process,Queue
import time
import random

def consumer(name,q):
    for i in range(5):
        res=q.get()  #从队列中提取一个元素,若队列为空则阻塞
        print(u"消费者%s 吃%s。" %(name,res))
        time.sleep(random.randint(1,3))
    print(name+'吃饱了。')

def producer(name,q):
```

```
        for i in range(5):
            time.sleep(random.randint(1,2))
            res=u'包子%s'%i
            q.put(name+res)  #把 re 插入队列
            print(u"生产者%s 生产了%s。" %(name,res))
        print('生产者%s 任务完成了。' %name)

    def consumer_producer():
        """ 生产者-消费者模型 """
        queue=Queue(2)#一个队列,长度为 2
        p1=Process(target=producer,args=('Tom',queue))
        p2=Process(target=producer,args=('Jerry',queue))
        c1=Process(target=consumer,args=('小明',queue))

        p1.start()
        p2.start()
        c1.start()
    consumer_producer()
```

运行结果：

```
生产者 Tom 生产了包子 0。
消费者小明吃 Tom 包子 0。
生产者 Jerry 生产了包子 0。
生产者 Jerry 生产了包子 1。
消费者小明吃 Jerry 包子 0。
生产者 Tom 生产了包子 1。
消费者小明吃 Jerry 包子 1。
生产者 Jerry 生产了包子 2。
消费者小明吃 Tom 包子 1。
生产者 Tom 生产了包子 2。
消费者小明吃 Jerry 包子 2。
生产者 Jerry 生产了包子 3。
小明吃饱了。
```

例 3-9 展示了主进程和子进程利用队列进行通信的过程。

共享内存用一个长度为 2 的队列来实现。如果队列满了，生产者被阻塞等待；如果队列空了，消费者被阻塞等待。

每个生产者负责生产 5 个包子，并把包子放入传送带(队列)，任务完成后下班退出。

每个消费者吃 5 个包子，每次从传送带(队列)拿包子，吃够 5 个停止。

这次任务有 2 位生产者 Tom 和 Jerry，有 1 位消费者小明。

运行结果显示，小明吃够了 5 个包子，“吃饱”走了；Tom 生产了 3 个包子，Jerry 生产了 4 个包子，二者停止了工作，但没有退出，这是因为二者合计生产了 7 个包子，被小明吃掉 5 个，还剩 2 个，但是传送带(队列)满了，两个生产者被阻塞。

这个例子中，3 个进程都未调用阻塞方法 join()，因此互不等待，各干各的。承担阻塞

和协调任务的是充当缓冲区的队列。

调整这段代码里的队列长度，调整生产者和消费者的数量，可以观察到不同的运行结果。例如队列长度改为 10，可以观察到如下结果：

```
生产者 Jerry 生产了包子 0。
消费者小明吃 Jerry 包子 0。
生产者 Tom 生产了包子 0。
生产者 Jerry 生产了包子 1。
消费者小明吃 Tom 包子 0。
生产者 Tom 生产了包子 1。
生产者 Jerry 生产了包子 2。
消费者小明吃 Jerry 包子 1。
生产者 Tom 生产了包子 2。
生产者 Jerry 生产了包子 3。
生产者 Tom 生产了包子 3。
生产者 Jerry 生产了包子 4。
生产者 Jerry 任务完成了。
消费者小明吃 Tom 包子 1。
生产者 Tom 生产了包子 4。
生产者 Tom 任务完成了。
消费者小明吃 Jerry 包子 2。
小明吃饱了。
```

3.3.4 进程异步

与进程同步(synchronous)对应的是进程异步(asynchronous)。在进程同步场景下，主进程作为调用者，需要等待子进程返回执行结果，为此需要通过各种阻塞方法协调进程间的执行顺序。进程异步时，调用者无须等待返回结果，继续执行后续的代码，而异步执行的进程通过状态、通知和回调等方式来通知调用者。显然，异步方式的并发效率相对较高。

使用 Python 的进程异步，一种简便易行的方式是使用 multiprocessing 模块内的进程池 Pool 类。

通过 Pool，只需一行代码就可以创建多个进程；对于包含多个任务的队列，特别是任务数多于进程数量的场景，可以通过 map()方法，让进程池内的进程在完成一次任务后，自动提取剩余的未处理过的任务。

创建 Pool 类对象后，可以调用异步执行或同步执行的方法，使用语句如下：

```
from multiprocessing import Pool
pool=Pool(n)            #创建 n 个进程
#异步,一次处理一个任务,多个任务时需多次调用
pool.apply_async(func,args=(x,),callback=None)
#同步
pool.apply(func,args=(x,))

#异步,处理多个任务
```

```
pool.map_async(func,iterable_args,callback=None)
#同步,处理多个任务
pool.map(func,iterable_args)

pool.close()          #关闭进程池,关闭后不再接收新的请求
pool.join()           #阻塞主进程
```

可以注意到,进程池用完之后需要调用 close()方法,停止接收新的任务请求,可以调用 join()方法阻塞主进程。

从进程池里摘下进程执行计算任务的方法中,异步方法主要有 apply_async()和 map_async()。其中,apply_async()一次处理一个任务,注意形参 args 传递数据时,一定要使用(x,)形式,逗号不能缺少,否则触发错误。map_async()适合批量处理一批计算任务,所以第二个形参通常是一个列表类型。

这两个方法支持下面两种返回执行结果方式。

(1) 返回 multiprocessing.pool.AsyncResult 类对象,可以通过调用对象方法 get([timeout])提取返回的内容。注意,get()方法有阻塞等待作用。

(2) 通过回调函数(callback)获得结果。所谓回调函数,从表现形式来看,是通过把函数名当作参数传递,从而实现函数的调用;在进程异步的场景里,主进程发起异步调用后,子进程调用主进程的函数,从而将结果返回给主进程,这样的函数就是回调函数。Python 的回调函数要求接收一个参数,并且执行效率要高,否则会阻塞处理结果的线程或进程。

【例 3-10】　进程异步与结果返回

```
from multiprocessing import Process, Pool
def calculator(x):
    print('线程计算...')
    return x * x * x

def callback_print(result):
    #callback 函数效率要高,否则线程会被阻塞
    print(f'回调函数,输出运行结果: {result}')

def async_pool():
    tasks=[1,2,3]
    pool=Pool(4)
    print('apply_async:')
    pool.apply_async(func=calculator,args=(tasks[0],),callback=callback_
print)
    pool.apply_async(func=calculator,args=(tasks[1],),callback=callback_
print)
    result=pool.apply_async(func=calculator,args=(tasks[2],))
    print('不用回调函数打印,result=',result.get(timeout=1)) #get()可阻塞进程
    print('map_async:')
    pool.map_async(calculator,tasks,callback=callback_print)
    pool.close()#关闭进程池,关闭后,不再接收新的请求
```

```
    pool.join() #必须调用,否则主进程先行结束退出,子进程无法回调

async_pool()
```

例 3-10 中,计算任务是一个仅完成三次方计算的函数 calculator();callback_print()是回调函数,用于打印输出传递给它的计算结果。

异步执行方法 apply_async()一次接收和处理一个数据,所以调用了 3 次以处理 3 个数据。其中前两次调用都通过回调函数返回结果,第 3 次调用时,用一个变量接收 apply_async()返回的对象,并用 get()方法提取出计算结果。

map_async()方法可以异步地批量处理任务,所以直接传递任务列表 tasks 给它,并用回调函数返回计算结果。结果如下:

```
apply_async:
线程计算...
线程计算...
回调函数,输出运行结果: 1
线程计算...
回调函数,输出运行结果: 8
不用回调函数打印,result=27
map_async:
线程计算...
线程计算...
线程计算...
回调函数,输出运行结果:[1, 8, 27]
```

可以看到 apply_async()返回了 3 个并发执行计算的结果,而 map_async()则是在完成任务后,系统把结果封装成一个列表返回。

需要说明,第 3 次 apply_async()调用后的打印语句,即 print('不用回调函数打印,result=',result.get(timeout=1)),对结果输出有影响。如果将这条语句注释,则结果如下:

```
apply_async:
map_async:
线程计算...
线程计算...
回调函数,输出运行结果: 1
线程计算...
回调函数,输出运行结果: 8
线程计算...
线程计算...
线程计算...
回调函数,输出运行结果:[1, 8, 27]
```

对比前后两次输出结果,发现“map_async:”这个输出信息被提前了。显然,在第一个情景中,方法 result.get()的调用使得主进程直到等待 apply_async()返回结果后,才执行

map_async()。

本节简要介绍了 Python 多进程的基本实现方法、进程间通信方法、进程的同步和异步。如果实际应用中不需要这些复杂的机制，可以考虑用线程来代替进程。

3.4 Python 多线程

线程是比进程更小的基本执行单位。通常来说，一个进程可以拥有多个线程，这些线程共享该进程内的所有资源。

有的操作系统(如 Linux)内核并不直接支持线程，而是由用户程序自行创建和管理线程，称为用户级线程；有的操作系统(如 Windows)在内核中支持线程，称为内核级线程，并把线程作为 CPU 调度的基本单位。也就是说，不同操作系统在线程的实现上有所不同，不同开发工具包在线程实现上也有所不同。

Python 的线程实现和管理使用了全局解释器锁(GIL)机制，本质上是串行执行的。在多核 CPU 上运行 Python 多线程时，可以观察到线程并没有把每个核都用满，而 Python 多进程可以做到把 CPU 跑满的效果。

那么，Python 多线程还有意义吗？毫无疑问是有的。最典型的应用场景就是处理 I/O。例如编写爬虫爬取网页，每次请求页面访问都是一次网络 I/O，程序需要等到收到服务器响应的信息后才会执行后续的代码。此时，用多进程或者多线程方式同时访问多个页面，可以提高网络 I/O 效率，不必因一次 I/O 的延迟而无谓等待。因为创建多线程的系统开销要显著小于多进程，所以这种应用场景下，通常采用多线程。

3.4.1 创建线程

创建 Python 线程，需要导入线程模块：

```
from threading import Thread
```

线程的创建方法与进程相同。例 3-11 演示了用线程计算从 0 加到 100。

【例 3-11】 线程创建

```
from threading import Thread

def counter(n):
    cnt=0;
    for i in range(n+1):
        cnt+=i;
    print( cnt)

def one_thread():
    #初始化一个线程对象,传入函数的 counter 参数
    th=Thread(target=counter, args=(100,));
    th.start();
    #主线程阻塞,等待子线程结束
```

```
    th.join();

one_thread()
```

3.4.2 线程间通信

还记得无法用全局变量在多进程间共享数据的例子吗？如果这个例子用线程实现，会得到什么结果呢？

【例 3-12】 利用全局变量通信

```
share_num=0        #全局变量

def worker_1():
    global share_num
    share_num+=20
    print(u"worker_1,share_num=%d" %share_num)

def worker_2():
    global share_num
    share_num+=100
    print(u"worker_2,share_num=%d" %share_num)

def thread_sharedata():
    t1=threading.Thread(target=worker_1)
    t2=threading.Thread(target=worker_2)
    t1.start()
    t1.join()
    print(u"t1 启动后,主进程 share_num=%d" %share_num)
    t2.start()
    t2.join()
    print(u"t2 启动后,主进程 share_num=%d" %share_num)

thread_sharedata()
```

运行结果：

```
worker_1,share_num=20
t1 启动后,主进程 share_num=20
worker_2,share_num=120
t2 启动后,主进程 share_num=120
```

对比多进程的运行结果，可以发现例 3-12 成功实现了数据共享。这是因为线程共享了所属进程的资源，即主进程的内存资源，其他线程都可以访问。同一个进程中的全局变量对各线程来说，只要变量名相同，就是同一个变量。

当然，实现进程同步和通信的机制，例如信号量、锁、管道、队列等，同样可以适用于线程。因此同步模型生产者-消费者同样可以用多线程实现。