# 生产者消费者模型

在并发编程中使用生产者和消费者模式能够解决绝大多数并发问题。该模式通过平衡生产线程和消费线程的工作能力来提高程序的整体处理数据的速度。

## 为什么要使用生产者和消费者模式

在线程世界里，生产者就是生产数据的线程，消费者就是消费数据的线程。在多线程开发当中，如果生产者处理速度很快，而消费者处理速度很慢，那么生产者就必须等待消费者处理完，才能继续生产数据。同样的道理，如果消费者的处理能力大于生产者，那么消费者就必须等待生产者。为了解决这个问题于是引入了生产者和消费者模式。

## 什么是生产者消费者模式

生产者消费者模式是通过一个容器来解决生产者和消费者的强耦合问题。生产者和消费者彼此之间不直接通讯，而通过阻塞队列来进行通讯，所以生产者生产完数据之后不用等待消费者处理，直接扔给阻塞队列，消费者不找生产者要数据，而是直接从阻塞队列里取，阻塞队列就相当于一个缓冲区，平衡了生产者和消费者的处理能力。

有些模块负责生产数据，这些数据由其他模块来负责处理(此处的模块可能是：函数、线程、进程等)。产生数据的模块称为生产者，而处理数据的模块称为消费者。在生产者与消费者之间的缓冲区称之为仓库。 可以说生产者负责往仓库运输商品，而消费者负责从仓库里取出商品，这就构成了生产者消费者模式。

此模型经常会在实际生产中遇到，有以下优点：

* 解耦
* 并发
* 支持忙闲不均 当生产者制造数据快的时候，消费者来不及处理，未处理的数据可以暂时存在缓冲区中，慢慢处理掉。而不至于因为消费者的性能造成数据丢失或影响生产者生产。

生产者消费者模型在 Python 中一般有 2 种实现：

* 多线程和队列
* 生成器 yield

## 多线程与队列实现

以经典的生产包子为例

# content of queue\_test.py
from threading import Thread
from time import sleep
from queue import Queue

class Producer(Thread):
 def \_\_init\_\_(self, worker, queue):
 super().\_\_init\_\_()
 self.\_worker = worker
 self.\_queue = queue

 def run(self):
 while True:
 if 0 <= self.\_queue.qsize() <= 10:
 queue.put('baozi')
 print('{} 生产了1个包子, 一共{}个包子'.format(self.\_worker,
 self.\_queue.qsize()))
 sleep(0.5)
 elif 10 < self.\_queue.qsize() <= 20:
 queue.put('baozi')
 print('{} 生产了1个包子, 一共{}个包子'.format(self.\_worker, self.\_queue.qsize()))
 sleep(1)
 else:
 print('仓库较多，生产者休息3秒钟。')
 sleep(3)

上面是生产者代码，Queue 为队列，作为仓库角色。接下来看消费者代码

# content of queue\_test.py
class Consumer(Thread):
 def \_\_init\_\_(self, client, queue):
 super().\_\_init\_\_()
 self.\_client = client
 self.\_queue = queue

 def run(self):
 while True:
 if self.\_queue.empty():
 print('仓库没有包子了。。。')
 sleep(0.5)
 else:
 result = self.\_queue.get()
 print('{} 消费了1个包子, 还剩{}个包子'.format(self.\_client, self.\_queue.qsize()))
 sleep(0.5)

queue = Queue(maxsize=20)

for item in ['LiBao', 'YangBao']:
 temp = Producer(item, queue)
 temp.start()

for item in ['ChengBaoConsumer', 'TianBaoConsumer']:
 temp = Consumer(item, queue)
 temp.start()

## yield实现

import time

def consumer(name):
 print('{}准备吃包子了！'.format(name))
 while True:
 baozi = yield #在它就收到内容的时候后就把内容传给baozi
 print('包子【{}】来了，被【{}】吃了'.format(baozi,name))

def producer():
 c1 = consumer('A') #它只是把c1变成一个生成器
 c2 = consumer('B')
 c1.\_\_next\_\_() #第一个next只是会走到yield然后停止
 c2.\_\_next\_\_()
 print('开始做包子了')
 for i in range(1,10):
 time.sleep(0.5)
 print('三秒做了两个包子')
 c1.send(i)
 c2.send(i+1)

producer()