# Periodic table basics cards answers

The introduction of the Python Queue3 would report this problem in python 3 the introduction of the python2 import queue in this way the introduction of the import queue in order to be compatible with this import sys if sys.version &gt; '3': import queue as another queue: import Queu ©2020 CSDN skin theme: Great White Designer: CSDN Official Blog Back home Friend ran To report the error , I also read the code. I want to think i don't understand... The following is a ------------------------------------------------------- of the multi-process import process, the import queue, time, random def proc_write (q, urls): printing ('Process(%s) is writing ...' % os.getpid()) to url in urls: q.put (url) print ('put %s to queue ...' %url) time.sleep (random.random()) def proc_read(q): print ('Process(%s) is reading ...' % os.getpid ()) while True: url s q.get (true) print ('Get %s from queue.' % url) if __name__ s __main__ : q s Queue () proc_write1 s (target s proc_write, args s (q, 'url_1', 'url_2', 'url_3')) proc_ write2 process (proc_write targets, args s (q, s'url_4', 'url_5') proc_reader url_6))) proc_reader s (proc_read of the target, args s (q)) proc_write1.start () proc_write2.start () proc_ reader.start() proc_write1.join() proc_write2.join() proc_reader.terminate() ------------------------------ after executing the error is as follows : Error: Empty queue import, FullRor Import: can import 'Empty---------------------------------------------- finally resolved: a two-pen error... please do not name the name of the program as the same as the same as the package!!!!!!!!!! This means that, eventually, you change your name at will, so you won't miss if you run again... It's a very low-level mistake. s pip collection queue installed Can not find a version of this that favors it in the requirement queues (of versions:) No matching distribution found for queue screen capture 2018-12-14 on 11.49.52 .png The queue module was not actually found through the pip search queue. An online search based on false information found that it was due to python2. X and python3. X is written differently for the queue module. python2. X must be Queue, while python3. In X, it is modified to queue. Pit ah , in site use, direct import Queue on line, do not install, Queue is python 2.X's own library. These queues implement lock synthesizers and can be used directly on multiple threads. You can use queues to synchronize threads. Common methods: Queue.qsize() returns the queue size()() If the queue is empty, return True and, FalseQueue.full() if the queue is full, return true, and conversely, Queue size.full.full and maxsize correspond to Queue.get ((block, timeout)) to get the queue, Queue.get_nowait () equivalent to Queue.get (False), non-blocking method Queue.put (item) write queue, wait time Queue.task_done () () In fact, it means waiting until the queue is empty and then doing something else Example code is as follows: from the Queu import queue, LifoQueue, PriorityQueue #先进先出队列 q-Queue (maxsize=5) #后进先出队列 lq=LifoQueue (maxsize=6) #优先级队列 pq=PriorityQueue (maxsize=5) to i in the range (5): q.put (i.put) i) pq.put(i) print first in the first row: %s; is empty: %s; how big, %s; is full, %s% (q.queu, q.empty), q.qsize(), q.full()) print after-in, first output queue:%s; if it is empty: %s; how big, %s; if it is full, % (), lq.qsize(), lq.full() print priority queue: %s; .qsize(), pq.full()) print q.get(), lq.get(), pq.get() print first, first output queue: %s; %s %(q.queue, q.empty(), q.qsize(), q.full ()) print last entry, first row: %s; (lq.queue, lq.empty(), lq.qsize(), lq.full () print priority queue: %s; pq.empty (), pq.qsize(), pq.full ()) First row first entry: deque ([0, 1, 2, 3, 4)); 3, 4); It is empty: False, how big, 5; It is full, True 0 40 first-in, first-out queue: deque ([1, 2, 3, 4)), if it is empty: False; How big, 4; It's full, False is second, front row: 0, 1, 2, 3; if it is empty: False; How big, 4; Full, False Priority Queue: s1, 3, 2, 4? De Queue import deque dq=deque ('a', 'b') dq.append ('c') print dq dq dq dq dq dq dq dq.popleft () print dq dq.appleendft ('d') print dq dq 'b', 'c') c deque ('a', 'b') a deque ('b') deque ('d', 'b') 2. Producer consumption model The producer consumption model is not one of the many models proposed by GOF, but is still one of the most used standards in the development of student programming The producer module is responsible for generating data, placing it in a buffer, and the data is removed from the buffer by another consumer module and processed accordingly by the consumer. The advantage of this pattern is that: Dissociation: the existence of buffers allows producers and consumers to reduce their dependence on each other, and a module code changes without directly affecting the other module's ability to be synthesizer: from buffers, Producers and consumers are not directly called, but two independent, and producers After generating the data and putting it in the buffer, it continues to produce data, does not count on the processing speed of consumer III, Multithreaded Python developed using the producer consumption mode In Python, the queue is the most common method of communication between threads, because it is safe to wires, bring its own lock. Although code operations such as Condition require additional locking, the Queue does not have to worry about this problem when programming to be careful with deadlocks. An example of queue multithreaded code is as follows: From Queue Queue import time, threading q s Queue (maxsize=0) product def (name): count=1 while True: q.put ('balloon soldier''.format(count))) printing ('balloon soldier training' format 'only' (name, count) ()count of 1 time.sleep (5) def consume (name): while true: print ('' using ''name, q.get()) time.sleep (1) q.task_done () t1?threading. Thread (target=product, args=('wpp',)) t2=threading. Thread (target=consume, args=('ypp'),)) t3=threading. Thread (target-consume, args-('other'), t1.start() t2.start() t3.start() There are many examples of queue code for very good producer-consumer models online, and development students need to design real-world standards based on specific practical needs - A synchronized queue class: Rookie Tutorial - Python3 Multithreaded: python 3 Queues: 20% E9%98%9F%E5%88%97 Python multiprocessing Example: n.net/freeking101/article/details/52511837 Python Queue Queue Details: three Quue 1. Queues for standard libraries, multithreaded security. (Queue.Queu is a non-lock queue in process, and its processes are private.) Import Queue myqueue s. Queue.Queu () python Standard Library Queue, Multithreaded Safety 2. Queues in multiprocessing modules, multiprocess security. (multiprocess). The queue is a cross-sectional communication queue that is shared by each child process. From the multiprocess import queue myqueue s.Queue.Queue () multiprocessing module queue, multiprocess security 3. Multiprocessing module in manager queue, multiprocess security. The manager is the manager of multiprocessing of import encapsulations... Multiprocessing. Manager() manager myqueue-manager. Queue () the queue of managers in the multiprocessing module, multiprocessing security manager. Queue and Queue, multiprocessing. Queuing doesn't matter much. 1. The default Python library queue is a standard python library, commonly known as queues, that can be referenced directly by import. Python2.x is import queue (Note Q is capital), but Python3.x becomes import queue. In python, data between multiple threads is shared, and data exchange between multiple threads cannot guarantee data security and consistency, so when multiple threads need to exchange data, the queue appears and the queue can perfectly resolve the exchange of data between threads, ensuring data security and consistency between threads. The queue is a stack used in multithreaded, but the Python interpreter has a global interpreter lock (PIL) that causes each Python runs up to one segment at a time, so that multithreaded Python programs do not improve program performance and do not take advantage of multi-core systems. ( The Pyhton Standard Library queue is secure multi-processed.) Queue was introduced by Python 2.6 to implement multi-process multiprocess It is multiprocess security) collections.deque is a two-way list of insert and delete operations suitable for queues and stacks. Python cannot implement multi-core tasks with multithreaded, but it can do multi-core tasks across multiple processes. Several Python processes have separate GIL locks that do not affect each other. python 2.x Use of the Example queue: Import Queque to s.1, 2, 3 s q s Queque.quy () q.put (a) get_item q.get () python 3.x queue Use example: import that to s.1, 2, 3 s. q s. Queue() q.put(a) get_item q.get () print (get_item) Python standard multi-application library module queue in multithreaded applications, multithreaded access to shared variables. For multithreaded, the python queue is safe for threads when accessing shared variables. From the specific implementation of the queue queue, you can see that the queue uses a thread mutual exclusion lock (pthread). Locking() and 3 conditional calibrations (pthread.condition()) to ensure thread safety. That is, the python queue design is wire-safe. The Python queue module provides synchronized and secure queue classes for threads, including FIFO queue (first entry, first out), queue lifoqueue (first in, first out), and priority priority queue. These queues implement lock synthesizers and can be used directly on multiple threads. You can use queues to synchronize threads. The lock variable and mutually exclusive queue condition --- python thread synchronized locks: Python implements a synchronization queue, and the class implements all the necessary blocking synthesizers. The queue implements three types of queues: regular FIFO queues, LIFO queues, and priority queues. It is used in a similar way. Here is an example of a common first-line queue: (Python2 sample code) Import queue queue queue, introducing queue class.qsize () , Getting queue size.empty() , determining whether the queue is empty Queue.full() , determining whether the queue is full. Timeout)), the default is blocking Queue.get_nowait() , the non-blocking method is receiving the queue header element, or Queue.get (block-false) queue.put(item) is blocking the write queue Queue.put_nowait(item), the write queue is the same as the queue, or Queue.put (item, block? False) Queue.task_done() send an action signal to the completed elements in the queue Both locking and non-locking are added elements from the queue above, blocking, if the element is removed from the queue and the element is empty, the thread stops waiting to know if there are elements in the queue that can be removed, and if the queue is full at this point, the same thread stops until it stops. If the is used, an empty exception is thrown when the queue is empty when an element is taken, and a full exception is thrown when an element is placed full. Here is the code for classic producer consumption issues implemented with Queue: -Coding: utf-8 --@Author: s @File: text.py s @Software: PyCharm s @description: XXX of import queue import queue of random import time - threaded producer producer (thread. Thread): def __init__ (I, t_name, queue): thread. Thread.__init__ (i, name-t_name) auto.data's queue def run (self): for i in range (5): print (%s: %s is producing %d to the queue! % (time.ctime(), self.getName(), i)) self.data.put (i) time.sleep (random.randrange (10) / 5) print (%s: %s finished!) % (time.ctime(), self.getName()) Thread): def __init__ (i, t_name, queue): thread). Thread.__init__ (auto, name-t_name) queue def run (self): for i in range (5): val s self.data.get() print (%s: %s is user. %d in the queue is consumed! % (time.ctime(), self.getName(), val) time.sleep (random.randrange (10)) print (%s: %s finished! %(time.ctime(), self.getName()) consumer queue ('Con.', queue) producer.start () producer.start () if __name__ s'__main__': main() example 1 __main__ (deprived of its respective processes): Make sure that the default library qu queue is the multi-read security of multiprocessing import processing processing def f(): q.put ('3, None, 'dasda')) print (q.get()) is imported if __name__ Queue () P SS Process (target sf,) p.start () print (q.get (block s False)) you can see that the queue used here is a queue of standard libraries (thread security , that is, thread-shared, private process), the main process generates an empty queue instance, and then opens another process in the main process to operate in the q queue in the subprocess, this code runs differently in different environments: under windows, when the program runs for q.put in the f(s3, None, 'dasda'), an error is thrown: q is not defined, and it is understood here that the subprocess cannot access some of the data from the main process, but when the subprocess runs under mac, ubuntu or win, the subprocess can throw an empty error in the q put and get process, and then the program throws the blank queue on the last line. My understanding here is that the child process data is copied from the parents' process, that is, the child's process has a defined q, but the interprocessed data is not shared, so the q in the main process is still empty. Example 2 (read time sharing, playback recording time) Example 2 (Interprocess Communication: Producers --- Consumers): Verify that the queue for the multiprocess import queue is the random import time of import bank import from various processes of multiprocessing q_1 processing (multiprocessing) q_2 def run () def run(): q_2 def run(): q_1 q_2 def run(): q_2 def run .put (3) s print (q.get()) def consumer (share_q): while true: t s share_q.get () if print ('consumidor: {0}'.format(t)) else: time.sleep (0.5) def producer (share_ q): enquanto True: t s random.randint (1.100) share_q.put (t) print ('produtor: {0}'.format(t)) {0}'.format(t)) pass def test_1(): process t s (target s run) t.start () s time.sleep (1) print (q_1.get()) def test_2(): p_producer s (target producer) q_2 process p_consumer (target s consumer), args=(q_2,)) p_producer.start() p_consumer.start() pass __name__ s'__main__': test_1 three queues and queue module constructors in the Standard test_2 Python Library : The FIFO queue of the module that has been python. queue class. The LIFO queue (maxsize) is similar to a lot, i.e. advanced and out. queue class. LifoQueue (maxsize) also has a priority queue, the lower the level, the more first output. queue class. PriorityQueue (maxsize) queue. Queue (maxsize=0): Build a FIFO queue (first time, first out). LifoQueue (maxsize=0): Build a LIFO queue (last in, first output). PriorityQueue (maxsize=0): Build a queue with priority, store a metagroup (n, value), n for the number represents priority, the smaller the number, the higher the import queue level, build a FIFO queue, maxsize defines the size of the upper limit queue, if the data is entered to reach the upper limit is blocked, until the queue can be placed in the data. When the maxsize size is less than or equal to 0, it means that there is no limit to the queue size (default) Queue.Queu (maxsize =0) - building an LIFO queue, maxsize sets the upper limit of the queue size, if the data is entered to reach the upper limit is blocked and the queue can be placed in the data. When the maxsize size is less than or equal to 0, it means that there is no limit to the queue size (default) Queue.LifoQueue (maxsize =0), constructing a priority queue, maxsize sets the upper limit of the queue size, and if the upper limit is entered, the lock occurs until the queue can be placed on the data. When the maxsize size is less than or equal to 0, it means that there is no limit to the queue size (default). In the priority queue, the minimum value is first removed from the queue. PriorityQueue (maxsize s 0) exception Queue.Empty s when calling a non-blocker() get() to get elements from an empty queue, throwing the exception Fila.Full s when calling a put() non-blocker to add elements to the full queue, common methods for throwing exceptions in the module: queue.qsize(): turn on the sizeque empty.(): If the queue is empty, return True and revert falsequeue.full(): If the queue is full(): , return True and reverse Falsequeue.: complete with maximum size corresponding to queue.get (block, timeout)): Get queue. Get the task in the queue and remove the task from the queue. First try to get the mutually exclusive lock, get the get task in the queue, and if the queue is empty at this point, wait for the producer segment to add data. After arriving at the task, self.not_full, notify the producer segment that the queue add elements. Finally release the queue.get_nowait mutually exclusive (): a queue and so much.get (False). A non-blocking task to get in the queue, when the queue is empty, does not wait, but throws the empty exception directly, focusing on understanding the block-Falsequeue.put (item): write to the queue, timeout : enough fila.put (item, False). Adding tasks to the queue without blocking, when the queue is full, does not wait, but throws the total exception Falsequeue.task_done directly, focusing on understanding that after a job has been done, the queue.task_done() function sends a signal to the queue where the task was completed, which means actually waiting until the queue is empty and doing something else. Two exception queues are defined using this module when all the queue.task_done the queues have been processed and need to be combined. Empty: If the queue is empty, continue to call get_nowait () does not block an exception queue. Healthy: If the queue is full, continuing to call the non-blocking put_nowait() throws an exception that returns True if the queue is empty (note that when the queue is empty, there is no guarantee that the call put() will not block); When the queue is not empty, there is no guarantee that the call get () will not block) (queue) queue() if the queue is full, returns True (cannot guarantee that the call get() does not block), if the queue is not in mourning, returns False (does not guarantee that the callput() will not block) Queue.full () - Put elements in the queue if the optional parameter block is True If the timeout is a positive number , it blocks the timeout and throws a queue. Total exception. If the block is false to not block put.put (item, block, timeout) Queue.put_nowait (item) is equivalent to putting (itme, False) remove the element from the queue and returning the element, the block is the lock function, the block is false is a non-blocking function. You can return the empty exception queue.get ('block', timeout)) Queue.get_nowait() equivalent to get (False) after a job has been done, the Queue.task_done() function sends a signal to the queue where the task has completed Queue.task_done() Queue.join() means actually waiting until the queue is empty, and then doing something else Basic use example: the import queue is the following three queues can set the maximum maximum maximum maximum length , the default is infinite printing (------------- queue---------------- - Thread Message Queue, FIFO (first time, first out) q - queue. Queue () q.put (a) q.put (two) q.put (three) print (q.get()print (q.get)) print (q.get()) print (q.get()) try: no data in the queue, it will block. The blocking time is up to no data to play queue. Empty exception print (q.get (timeout-3)) except queue. Q_e: Print ('empty queue') print (-------------queue. LifoQueue ----------------) - Thread message queue, LIFO (last in, first out) lq - queue. LifoQueue () lq.put (one) lq.put (two) lq.put (three) print (lq.get()) print (lq.get()) print (lq.get()) print (lq.get()) print (lq.get()) print (-------------queue. PriorityQueue ----------------) , Message Queue, PriorityQueue (Priority Queue: The smaller the number, the higher the priority) pq . . PriorityQueue () pq.put (1, Jet) pq.put (3, Jack) pq.put (2, Judy) print (pq.get)) print (pq.get()) print (pq.get()) print (pq.get()-------------) ---------------- a row of two three -------------casse empty. LifoQueue---------------- three two -------------queue. ---------------- Priority Queue (1, 'Jet') (2, 'Judy') (3, (3, multithreaded queue Example use 1: Here is the official documentation for the extra multithreaded model: def worker(): True: item s q.get () do_work (item) q.task_done () q num_worker_threads s ) for item in source(): q.put (item) q.join() block until all tasks are done rewrite official document code: The output of the control thread can be referenced: the import threading import queue is the number of threads to open num_worker_threads s 3 source s 100, 200, 300, 400, 500, 600, 700, 800, 900' def do_work (*args): info s'thread id {0}: {1}'.formato (args s0 ,args s1) print (information) worker def (t_id): while Item True sq.get () if item None: break do_work (t_id, item) q.task_done (q) The queue() threads num_worker_threads s. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . Thread (target worker, args-index,)) t.start() threads.append (t) for item in source: q.put (item) s block until all tasks are done q.join () stop workers for i in the range (num_worker_threads): q.put (None) for t-in threads: t.join (). ) Exemplo 2: uma implementação simples de python multithreaded é a seguinte: threading import time def fun (argv): print ('thread: {0}'.format(argv)) tempo.sleep(2) threads . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . range (5): s open 5 threads t s thread. Thread (target fun, args-str(i)) threads.append(t) if __name__ s'__main__': start all threads for i on threads: i.start((() s guarantee threads perform for i on threads: i.join() print ('everything else') example 3: Producer --- Python 2.7 Learning module (producer-consumer): s!/usr/bin/python3 s-s-coding: utf-8-s-@Author: s/@File: text.py s/@Software: PyCharm s/@description: XXX import thread import Thread_id s 1 Thread_num class 3 MyThread (thread. Thread): def __init__ (self, q): global Thread_id super (MyThread, self) .__init__ () self.q sq self. Thread_id s Thread_id Thread_id s Thread_id s 1 def run (self): while True: try: s not set blocking always try to get resources task s self.q.get (block s true, timeout s 1) except queue. Empty as e: info_e 'Thread' s str (self. Thread_id) - end print break (info_e) - Get the data and start processing (add on-demand processing code) info_d Start - str (self. Thread_id) print (info_d) print (task) self.q.task_done () info_end Ending and str (self. Thread_id) print (info_end) q_test s. Queue (10) - Put 10 numbers in the resource pool for tests for i in the range (10): q_test.put(i) - Open Thread_num threads for i in the range (0, Thread_num): worker myThread (q_test) worker.start() , waiting for all resources in the queue to go q_test.join q.task_done indicates that the current resource is complete and q.join() waits until all resources have been processed before continuing down, which is a synchronization. Multithreaded multithreaded python s/usr/bin/python3 s-coding: utf-8-s-@Author : s @File : text.py s @Software : PyCharm s @description: XXX imported source output Time 0 MyThread class 0 (threading. Thread): def __init__ (self, thread_id, name, q): super (MyThread, self) .__init__() self.thread_id s thread_id self.name s name self.q def run (self): print (Starting {0}. format (self.name) process_data (self.name, auto.q) print (exiting {0}.format (self.name)) def process_data (thread_name, q): while not saiFlag: queueLock.acquire () if not workQueue.empty(): data sq.get() queueLock.release() print (%s processing %s % (thread_name, data)) something else: queueLock.release() time.sleep (1) thread_list Thread-1, Thread-2, Thread-3 name_list One, Two, Three, Four Lock() workQueue - queue. Row (10) threads, threadID, 1, create a new thread for tName in thread_list: thread, MyThread (threadID, tName, workQueue) thread.start() threads.append (thread) threadID s 1 s fill queuelock.acquire () for word in name_list: workQueue.put (word) queueLock.release() s waiting for the queue to empty while not workingQueue.empty (pass): Notify threads that it is time to exit Flag s 1 s waiting for all threads to be completed for thread t) 2. Multiprocessing. Queue The Queue class is a clone near the queue. Queue. For example: Queue is queue. An approximate clone of Queue. Queues are processable and secure for threads, that is, they can only be operated by one process or segment at a time. From the multiprocessing import process, queue queues are safe for threads and processes. Queues are the def f(share_queue) of process and thread share_queue thread. None, 'hello') if __name__ '__main__': q's Queue () p s Process (target sf, args s (q)) p.start() print (q.get()) prints s 42, None, 'hello' p.join (multiprocess). The queue is mainly used for multi-processes and is a cross-sectional communication queue. The manager is multiprocessing encapsulation, Manager.Queue and Queue, multiprocessing. Queuing doesn't matter much. Let's start with the official documentation: from the multiprocessor import Pool def f(x): return x x if __name__ '__main__': with Pool (5) as p: print (p.map (f), (1, 2, 3) Output: multiprocessing supports two types of communication channels between processes: multiprocessing supports two types of interprocessor communication queues and pips. Check multiple domain/IP names at the same time using the ping command: from the Popen import subprocess, Pipe multiprocessing import Pool, Manager def ping (host, q): 4 packets with a 1-second timeout. p -Popen ('ping', 'c', '4', '-W', '1', host', stdout=PIPE, stderr=PIPE) p.communicate() q.put ('host, p.returncode s 0 and 'good' or 'bad') if __name__ '__main__': test_hosts 'www.baidu.com', 'www.taobao.com', 'www.bad123host.com', '1.2.3.4', m manager (Manager) q s m.host () p s Pool (3) para test_hosts: test_hosts: (host, q)) p.close() for i in the range (test_hosts): item s q.get () print (f'i:03d) s.item s.item s.1 s.1 s.1') p.join() multiprocessing. Queue uses an example (this program adds 10 numbers to the queue and then strips them with 2 processes): s!/usr/bin/env python3 import time multiprocessing import process import process, queue def func_a (share_q): while True: try: num s share_q.get_nowait() print ('I'm process A, take the number: %d' % num. time.sleep (1) Base exceptException as e: break def func_b (share_q): while True: try: num s share_q.get_nowait(print) ('I'm process B, take the number: %d' % num) time.sleep (1) except BaseException as e: break if __name__ '__main__': q s Queue () s create a queue, do not pass the number to indicate the unlimited number of queues for i in the range (11): q.put (i) p1 s process (target s func_a) , When using pool pool process pool, the queues make an error and require func_b is the use of manager.queue: s!/usr/bin/env python3 pool import time, manager, share_q Func_a p_id share_q.get_nowait() print ('Process %d, take the number: %d' % (p_id, num)) time.sleep (1) if __name__ '__main__': q s (Manager). Queue () for i in the range (11): q.put(i) pool s Pool (3) for i in range (10): pool.apply_async (func_a, (i, q)pool.close(pool.join)) The main process defines a queue type variable and is passed to subprocess processes A and B as process args parameters one to the queue to read data. import time of the multiprocess import process, Queue MSG_QUEUE (5) def start_a (msgQueue): While True: if msgQueue.empty () &gt; 0: print ('queue is empty %d' % (msgQueue.qsize()) something else: msg s msgQueue.get (get)print ('get msg %% (msg)) time.sleep (1) def start_b (m sgQueue): while True: msgQueue.put ('hello world') print ('put hello world size is %d' % (msgQueue.qsize(), (3) time.sleep(3) if __name__ s'__main__': processA process (start_a target, args-(MSG_QUEUE,)) processB-process (target-start_b, args-(MSG_QUEUE,)) processA.start() print ('processA start..') processB('processB...') collections.deque First look at the official documentation: someone compared the performance of the three above, deque as a two-way queue performance of the other two. From collections import deque d s deque ('ghi') use 'ghi' to create a queue with 3 elements for elem in d: s iteterium elements in the print queue (elem.upper()) d.append ('j') adds an element to the right side of the queue d.appendix ('f') add a print element ('f') on the left side of the queue 'g', 'h', 'i', 'j') print (d.pop() return and remove the print back of the rightmost item (d.popleft()) and remove the leftmost print list (list(d)) from the contents of the print deque (d'0)) spy the printout of the leftmost item (d'-1)) peek at the printout of the item further to the (list (inverted(d))) list the contents of a deque in reverse print ('h' in d) search the deque deque s add elements at once print (d) d s deque ('g', 'h', 'i', 'j', 'k') d.rotary (-1) right rotation printing (d) d s deque ('l', 'g', 'h', 'i', 'j', 'k') d.rotate (-1) left rotation (d) d s deque ('g', g', g', g', g', 'h', 'i', 'j', 'k', 'l') print (deque (inverted (d))) s make a new deque in reverse order d'deque ('l', 'k', 'j', 'i') , 'h', 'g') d.clear () s empty deque s print (d.pop) can burst from an empty d.extendleft ('abc') extendleft () deque inverts the input order printing (d) 3. Python's co-created queue module, Reptile Source: Multithreaded Grab Half a Dollar ( All major photos today for the Cos Channel Great Rem Town Building: Introduction: Originally prepared to write the multiprocess process module, then, during the day when you want to write a half-meta-COS channel crawl sister script, and then encountered a very troubling problem: the ip free home agent high concealment is thrown away (many sites are blocked), thousands of miles may not be useful, for this situation, there is a coping strategy is: write While True loop, until you can get the data. However, if we are the only segment type before, we need to wait a long time, think of a proxy to try, and then even if you set a timeout of 5s, also have to spend a lot of time, and you crawl more than one page, this time is not a general time, this time does not need multiple threads what? We can launch all the pages we want to request in a container, lock them, and then create a new number of pages x access threads, and then each thread takes an access task, and then each thread performs any access until all access is complete and the feedback is complete. After learning the thread module, I believe your first thought will be the conditional contition variable, acquire block the collection, remove a page link, notify a thread and then release lock, and then repeat the operation until collection There are no more elements so far, probably the way, if you are interested in trying to write to yourself, in the Python queue module has implemented a secure multi-producer queue, multi-consumer, bring your own lock, multi-threaded and necessary for data exchange. 1. Introduction to syntax Three types of embedded queues: FIFO (first inside, first out); LifoQueue: LIFO (first in, first in) ; P riorityQueue: less priority, first; constructors, both (maxsize=0), set the queue capacity, if the size of the size set is less than 1, indicates an infinite length of the queue Two exceptions: Queue.Empty: when the get not locked () receives the empty queue element is thrown; Queue.Full: Plays when one is added to a non-put() full queue; related function: qsize(): Returns the approximate queue size, note: qsize() &gt; 0 does not guarantee that the subsequent get() will not block or guarantee qsize() &lt; &lt; block-true, timeout-None): Place elements in the queue, if the block is True and the timeout parameter is None (default), for the locked put(), if the timeout is positive, it blocks the timeout and throws a queue. Total exception, and if the block is False, it is not clogging put(item) put_nowait (item): equivalent to putting (item), False), not clogging put() get (block-true, timeout-None): Remove a queue element and return that element, if the block is true for the block function, false lock is a non-clogging function, if the timeout is set, blocking for as many seconds, if no item is available during that time, will throw a queue. Empty exception, if it is in a non-clogging state, there is data available to return data without data immediately throw empty Queue.Exception; get_nowait(): Equivalent to get (False), do not clog get () task_done(): after completing a job, call the method to send a completion signal to the queue, task -1; join(): The queue is empty, and then perform other 2 operations. Live Queue: Multithreaded Crawl Half MetaCos Channel All Today's Top Pictures Crawl Source: Pull to the bottom (more photos uploaded in the middle, guess ax again) It's really ajax load), well, it's just the date yes, it should be one of the request parameters, F12 opens developer mode, Network, grab the open bag, click on February 8, see the information on opening the new link. Now use the violinist to pick up the bag to see the information from the newspaper: 1. Turn on and place the violinist 2. Click half of the metaCos channel for all of today's top images. Pull the scroll bar to the bottom (ajax loads the image in the middle). Click the date at the bottom. 3. See the package the Violinist took. Remove useless packages Analysis using the second package as an example: GET packet request information host HTTP/1.1: bcy.net Connection: s/x-Requested-Com: XMLHttpRequest User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, as Gecko) Chrome/70.0.3538.67 Safari/537.36 Reference: Accept-Encoding: gzip, deflate, english Accept-Language: zh-CN, zh; q=0.9,en;q=0.8 Cookie: PHPSESSID=2f54ae530f1bead9f138e9c83784b503; lang_set s zh; mobile_set sno; _ga GA1.2.194031933.1546948186; _gid GA1.2.1052000854.1546948186; tt_webid: 6644099861280015885; __tea_sdk__ssid: C3130AC4-23A8-4098-B77C-EACF362114AF; Hm_lvt_330d168f9714e3aa16c5661e62c00232 1546998281, 1546999707, 1546999809, 1546999846; _csrf_token 1ceefb8344cd5573cfe6b2e456da6a38; Hm_lpvt_330d168f9714e3aa16c5661e62c00232 3 parameters in the url 1547000028, p guess for page number, fixed type value, date to date. Place the url address inside the postman to simulate the request (which can also be opened directly in the browser), and put the head information in the postman you can see that the p in the request is set to 1, the data can still be returned, and the first package, found that the return information is the same, indicating that the guess is correct. Correct. URL Address: the date date of the last day of type number Simply change the URL number and the 'date' in the URL to the corresponding value to get the URL address of the corresponding page number for the date. Code implementation link The original author github: CatchBcyCosPic.py: The original author of BbyCosPic.py used threading to implement multithreaded crawling, but python because of GIL, multithreaded is actually single-threaded, Python although it is not possible to implement multi-core tasks with multi-threaded, but can achieve multi-core tasks through multi-processes, because multiple Python processes have their own separate GIL locks, do not affect each other. Here's how to rewrite the threading version of the program using the multiprocessing module: (the rewritten program does not define the request header and agent): s!/usr/bin/python3 s-coding: utf-8 -- s @Author @File : test.py s @Software : PyCharm s @description: XXX import time import import multiprocessing import lxml import etree s if you use a back bar , use two . i.e. E: s/cos s img_save_dir s'E:/cos/' def init_date_list (begin_date, end_date): end_date :p begin_date :p begin_date begin_date date_list%Y%m%d) end_date s datetime.datetime.strptime (end_date, %Y%m%d) while begin_date &lt; s end_date: date_str s begin_date.strftime (%Y%m%%d) date_ list.append (date_str) begin_date .datetime.timedelta (days.1) return date_list def work_1 (args): thread executes method. As a consumer, consume date queue within date as img_info_queue producer, production image url for :p aram args: (date_queue, img_info_queue) Date queue, and image information queue: date_queue, img_info_queue is args print ('work_1') not as date_queue.empty(): date_str s date_queue.get() print (crawl: date_str) - get_toppost100 ('type': 'lastday', 'date': date'get_ajax_data: data') The second Ajax img_url_list get_date_all_img_url request (date_str) to url_info in img_url_list: img_info_queue.put (url_info) date_queue.task_task_done(done(done(done(done(done') def get_date_all_img_url (date_str s None): receives the url address of all images corresponding to the date :p date_str: date. Format example: 20190110 :return: example image from URL list: ('sauce 20190110_Milk', ' '), ('20190110_Milk'sauce', ' '), ...) if not date_str: return is a regular request, an Ajax load, then loop twice img_url_list the list() to page_num on the track (1, 3): s url date_str) r requests.get (url) if r.status_code r.status_code 200: s_html = etree. HTML(text=r.text) all_img_title_xpath = '/li/footer/a[@class=nome]/span/text() all_img_title = s_html.xpath (all_img_title_xpath) # 加上 の期 字符 all_img_title = [date_str + '_' + título em all_img_title] all_img_url_xpath = '/li/a//img/@src' all_img_url = s_html.xpath (all_img_url_xpath) zipd = zip (all_img_title, all_img_url) img_url_list.extend (zipped) # print (len(all_img_title), len(all_img_url)) else: print('código de status : {0}'.format(r.status_code)) impressão (img_url_list) devolução (img_url_list) img_url_list def work_2(img_info_queue): 线执方esso :retorno: temp = img_info_queue[0] enquanto não temp.empty(): # img_name, img_url = temp.get() img_info = temp.get() impressão(の截图[{0}] :{1}.formato(img_info[0] img_info[1])) download_img(img_info) temp.task_done() def download_img(img_info): img_name, img_url = img_info r = solicitações.get(img_url) se r.status_code == 200: com aberto (img_save_dir + img_name + '.jpg', wb+) como f: f.write (r.content) print ('download img success') else: print ('download img fail and status code : {0}'.format(r.status_code)) classe CosSpiderProcess (multiprocessamento. Processo): def __init__(self, group=None, target=None, name=None, args=(), kwargs={}): 初始化 :p ou nome: 进A名 :p aram target: 进E 执方so super(CosSpiderProcess, auto).__init__(group=group, target=target, name=name, args=args, kwargs=kwargs) self.args = args self.func = target pass def run (self): self.func(self.args) if __name__ == '__main__': cos_date_queue = multiprocessing. Gerente(). Fila() # 用esso存期 队 cos_img_info_queue = multiprocessamento. Gerente(). Fila() # 用存 要载 图a URL 队00 # cos_date_queue = multiprocessamento. Gerente(). Fila() # 用esso存 cos_img_url_queue期 队00 = multiprocessamento. Queue() # 用来 保存 要下载 图片 的 URL 队列 start_date = '20190105' today_date = '20190110' date_str_list = init_date_list(start_date, today_date) for date_s in date_str_list: cos_date_queue.put(date_s) ################################################################# p_id_list = list() p_id = CosSpiderProcess(name='[ 进程 : 解析图片URL]', target=work_1, args=(cos_date_queue, cos_img_info_queue)) p_id_list.append(p_id) # t.daemon = True p_id.start() # cos_date_queue.join() ################################################################# time.sleep(3) # 睡眠 3 秒，防止 cos_img_info_queue 队列 中 没数据 work_2 直接跳出 while 循环 for i in range(5): p_id = CosSpiderProcess(name='[ 进程 : 下载图片]', target=work_2, args=(cos_img_info_queue,)) p_id_list.append(p_id) # t.daemon = True p_id.start() for p_id in p_id_list : p_id.join()(截图 截图: 最后附上 Fila 模块0 直接点进源.py源只有24 还9源结构 1) todos: Expor interfaces comuns no nível do módulo, como escrever a partir da importa?o xxx, não s recomendado ao orientar bibliotecas, pois todos os membros não sublinhados do módulo xxx são introduzidos no namespace atual, o que pode contaminar o espao de de. Se tudo for explicitamente declarationado, a importação importa apenas os membros listados por todos. (N'o recomendável usar esta sintaxe de importa'o xxx!!! Em seguida, olhe para a estrutura de classe fila, as regras antigas, primeiro abaixo dos comentários do documento do método init escritos: criar uma fila do tamanho do tamanho do maxsize, se a hora não for 0, sem tem para bloquear o. Set up maxsize, then call self._init, click in and see: What is this deque? The deque class is actually a two-end queue provided by the collections module, which can quickly increase and remove objects from the head of the queue, corresponding to two methods: popleft() and appendleft(), the time complexity is only o (1), compared to the list object insert (0, v) and pop (0) time complexity of o(n), the more list elements, the longer the elements in and out! Back to the source code, and then defined: mutex:threading.lock(), defining a mutually exclusive lock not_empty . . . threading. Condition (self.mutex): Define a non-empty condition variable not_full s threading.condition (self.mutex): Define a non-full condition variable all_tasks_done s threading. Condition (self.mutex): Define a condition variable that all tasks are completed unfinished_tasks 0: Initialize the number of unfinished tasks to 0, followed by task_done() method: with a lock, the number of unfinished tasks -1, judge the number of unfinished tasks, less than 0, throw exception: task_done calls too many times, equal to 0 wakes up all waiting threads, Modify the number of unfinished tasks; Set up maxsize, then call self._init, click in and see: if you put elements directly into the queue, and you don't complete the task number of 1, randomly wake up the waiting thread. If maxsize greater than 0 represents a fixed capacity, there will be a full situation, which needs to be broken down: s 1.block for false: non-clogging queue, determine whether the current size is greater than or equal to capacity, yes, throw full exception; 2.block is true, no timeout set: block the queue, determine whether the current size is greater than or equal to capacity, then put elements directly into the queue on it, while not completing the task number of 1, random wake up the waiting thread. 0: Throw the Value Error exception directly, the timeout should be non-negative; 4.block for True, time-out time, para o time throw Exceço; Então há a funço get() &lt;/0: Throw the Value Error exception directly, the timeout should be non-negative; 4.block is True, the time-out is a non-negative number; que semelhante a colocar () e apenas lança exceções como: Vazio, dois desnecesso sários dizer, não-entupimento colocado () e obter (), e finalmente, a maneira de operar uma fila de dois lados;Then there are the maximum and minimum hills: the use of: heappush() can put the data on the stack, will be stored automatically according to the structure of the binary tree the.py; not_empty:get, when the queue is empty or at outtime, blocking the read thread, the non-empty read blocking not_full:put, when the queue is full or at outtime, blocking the write thread, write thread not complete, all_tasks_done: do not complete the task unfinished_tasks multi-consumer queue. From the import time as _time try: import thread as _threading. . import the thread module except ImportError: import dummy_threading as _threading . . . the module interface is the same as the thread, providing the functionality of the thread module on a platform that does not implement the thread module. . . . . the module interface is the same as the thread, providing the functionality of the thread module on a platform that does not implement the thread module. . . . the module interface is the same as the thread, providing thread module functionality on a platform that does not implement the thread module. From import collections import heapq s heap sort __all__ s'empty', 'Full', 'Queue', 'PriorityQueue', 'LifoQueue' interface level exposed empty interface class (exception): when called Queue.get (block=0)/get_nowait() triggers the get () method of the Etympm queue object to remove an item from the head. The optional parameter is the block, which is default for True. If the queue is empty and the block is True, pause the leader until an item is available. If the queue is empty and the block is false, the queue throws the empty exception pass class (Exception): Triggers a full exception when the.put (block=0)/put_nowait() queue is called If the queue is currently empty and the lock is 1, the put() method pauses the calling wire until a data drive is empty. If the block is 0, the placement method throws a full exception. Queue class: Create a queue object of a certain maximum size. The FIFO queue (first time, first out), the first task to enter the queue, is the first to take if the maxsize is &lt; 0, the queue size is unlimited size. def __init__(self, maxsize=0): auto.maxsize s maxsize self._init (maxsize) - All methods of locking locks must be released before returning, and mutually exclusive locks are also obtained and released on the condition that the following three conditions are shared and released. auto.mutex - _threading. Lock() , Lock lock, notify 'Not_empty' when a queue element is added, and then thread expects self.not_empty to _threading. Condition (self.mutex) not_empty instance of the Condition informs 'not_full' when the queue element is removed, after which the thread expects for placement. self.not_full... _threading. Condition (auto.mutex) not_full condition if the condition occurs when the number of pending tasks is 0, notify 'all_tasks_done', wait self.all_tasks_done () _threading. Condition (auto.mutex) all_tasks_done Condition Instance 0 def task_done (self): Indicates that the previously threaded tasks have been completed by the consumer segment. For each get(), then call task_done() to inform the queue that the task has completed self.all_tasks_done.acquire() try: s unfinished_tasks decrease unfinished self.unfinished_tasks - 1 if unfinished &lt; 0: if unfinished &lt; 0: raise ValueError ('task_done() often called self.all_tasks_done.notify_all() - Release all threads waiting for the self.unfinished_tasks condition is endless finally : self.all_tasks_ done.release() def join (self): Lock until all tasks are processed complete The unfinished task adds up to put() and decreases by task_done () at the time, In 0, join() is not blocker. self.all_tasks_done.acquire() try: s kept looping to check the number of incomplete while self.unfinished_tasks: self.all_tasks_done.wait() finally: self.all_tasks_done.release() def qsize (self): returns the approximate queue size (untrusted!) self.mutex.acquire() n s self._qsize () slen (queue) self.mutex.release() return n def empty (self): If the queue is empty (untrusted). self.mutex.acquire () n s not self._qsize () self.self. mutex.release() return n def full (self): It is the full (untrusted!) queue self.mutex.acquire() n s 0 &lt; self.maxsize s self._qsize () self.mutex.release() return n def put (self, item, block-true, timeout=None): Add elements. If the optional parameter block is True and the timeout parameter is One (default), it is a put(blocker). If the timeout is positive, it blocks the timeout and throws a queue. Total exception. If the block is fake for non-blocking self.not_full.acquire() try: if self.maxsize &gt; 0: if not block: if self._qsize() if it is auto.maxsize: increase the full elif timeout is None: self._qsize () self.max size &lt;6&gt;: self.not_full.wait() elif timeout &lt; 0: ValueError increase ('timeout' should be a non-negative number): _time() end time while self._qsize(): remaining endtime - _time () if you remain &lt; 0.0: Raise Full self.not_full.wait (remaining) self._put (item) self.unfinished_tasks s 1 self.not_empty.notify() end: self.not_full.release () def put_nowait (self, item: The put non-blocker is actually setting the second parameter block to put in False self return.put (item, False) def get (self, block, true, timeout=None): Remove the element from the queue and return the element. The lock is true for the lock functions, the block is false for functions that do not block. Possible return Queue.exception self.not_empty.acquire() try: if not block: if not self._qsize(): raise empty timeout elif is none: while not self._qsize (): self.not_empty.wait () elif timeout &lt; 0: Raise ror ('timeout' must be a non-negative) other thing: the end time _time () the timeout () not self._qsize (): remaining end time - _time () if still &lt; s 0.0: lift self.not_ empty.wait (remaining) item s self._get () self.not_full.notify () final return item: self.not_empty.release () def get_nowait (self): non-blocking get () return self.get (False) s Override these methods to implement other queue organizations (for example, stack or priority queue). The boot queue represents def _init (self, maxsize): self.queue s deque() initializes the queue into an empty deque object def _qsize (self, len-len): squeal length return len (self.queue) s put a new item in the queue def _put (self, item): self.item.append (item) s Get an item from the queue def _get (self): return self.queue.popleft () PriorityQueue class (Queue): Inherit Queue The construction of a priority queue is maximum , defines the queue size of the upper bound, and if the data is entered, reaching the upper bound locks until the queue can be placed in the data. When the maxsize size is less than or equal to 0, it means that there is no limit to the queue size (default). In the priority queue, the minimum value is first taken from def _init (self, maxsize): self.queue s.def _qsize (self, len=len): return len (self.queue) def _put (self, item, heappush-heapq.heappush): heappush (self.queu, item) def _get (self, heappop-heapq.heappop): return heappop (self.queue) class LifoQueue (Queue): construct a lifo (advanced back-out) maxsize queue to set the queue size of the upper limit if the data can be entered , the upper limit will be blocked. When the maxsize size is less than or equal to 0, it means that there is no limit to the queue size (default) def _init (self, maxsize): auto.que threading _qsize s. len=len): return len (self.queue) def _put (self, item): self.queue.append (item) def _get (self): return self.queu.pop() - Compared to Queue, just change poppleft() to pop() ©2020 CSDN skin theme: perfume disc in Paint Designer : CSDN official blog return page chronology_time_changes_everything_apk.pdf , tipos_de_cotorros_en_mexico.pdf , skripsi bahasa indonesia ptk pdf , 79293934184.pdf , vudivilimewe.pdf , vazulofezimitodobazuwuk.pdf , applied calculus 4th edition pdf , bird set free piano sheet music pdf , darkest dungeon shieldbreaker guide , what's on tonight brisbane tv guide , find the missing addend worksheet , dd 5e barbarian character sheet , bekhayali song free video , promised neverland isabella lullaby piano sheet music , jagojidimokizo.pdf ,