# 12.3 线程间通信¶

## 问题¶

你的程序中有多个线程，你需要在这些线程之间安全地交换信息或数据

## 解决方案¶

从一个线程向另一个线程发送数据最安全的方式可能就是使用 queue 库中的队列了。创建一个被多个线程共享的 Queue 对象，这些线程通过使用 put() 和 get() 操作来向队列中添加或者删除元素。 例如：

```
from queue import Queue
from threading import Thread

# A thread that produces data
def producer(out_q):
    while True:
        # Produce some data
        ...
        out_q.put(data)

# A thread that consumes data
def consumer(in_q):
    while True:
# Get some data
        data = in_q.get()
        # Process the data
        ...

# Create the shared queue and launch both threads
q = Queue()
t1 = Thread(target=consumer, args=(q,))
t2 = Thread(target=producer, args=(q,))
t1.start()
t2.start()
```

Queue 对象已经包含了必要的锁，所以你可以通过它在多个线程间多安全地共享数据。 当使用队列时，协调生产者和消费者的关闭问题可能会有一些麻烦。一个通用的解决方法是在队列中放置一个特殊的值，当消费者读到这个值的时候，终止执行。例如：

```
from queue import Queue
from threading import Thread

# Object that signals shutdown
_sentinel = object()

# A thread that produces data
def producer(out_q):
    while running:
        # Produce some data
        ...
        out_q.put(data)

    # Put the sentinel on the queue to indicate completion
    out_q.put(_sentinel)

# A thread that consumes data
def consumer(in_q):
    while True:
        # Get some data
        data = in_q.get()

        # Check for termination
        if data is _sentinel:
            in_q.put(_sentinel)
            break
```

```
        # Process the data
        ...
```

本例中有一个特殊的地方：消费者在读到这个特殊值之后立即又把它放回到队列中，将之传递下去。这样，所有监听这个队列的消费者线程就可以全部关闭了。 尽管队列是最常见的线程间通信机制，但是仍然可以自己通过创建自己的数据结构并添加所需的锁和同步机制来实现线程间通信。最常见的方法是使用 `Condition` 变量来包装你的数据结构。下边这个例子演示了如何创建一个线程安全的优先级队列，如同1.5节中介绍的那样。

```
import heapq
import threading

class PriorityQueue:
    def __init__(self):
        self._queue = []
        self._count = 0
        self._cv = threading.Condition()
    def put(self, item, priority):
        with self._cv:
            heapq.heappush(self._queue, (-priority, self._count, item))
            self._count += 1
            self._cv.notify()

    def get(self):
        with self._cv:
            while len(self._queue) == 0:
                self._cv.wait()
            return heapq.heappop(self._queue)[-1]
```

使用队列来进行线程间通信是一个单向、不确定的过程。通常情况下，你没有办法知道接收数据的线程是什么时候接收到的数据并开始工作的。不过队列对象提供一些基本完成的特性，比如下边这个例子中的 `task_done()` 和 `join()`：

```
from queue import Queue
from threading import Thread

# A thread that produces data
def producer(out_q):
    while running:
        # Produce some data
        ...
        out_q.put(data)

# A thread that consumes data
def consumer(in_q):
    while True:
        # Get some data
        data = in_q.get()

        # Process the data
        ...
        # Indicate completion
        in_q.task_done()

# Create the shared queue and launch both threads
q = Queue()
t1 = Thread(target=consumer, args=(q,))
t2 = Thread(target=producer, args=(q,))
t1.start()
t2.start()

# Wait for all produced items to be consumed
q.join()
```

如果一个线程需要在一个“消费者”线程处理完特定的数据项时立即得到通知，你可以把要发送的数据和一个 `Event` 放到一起使用，这样“生产者”就可以通过这个Event对象来监测处理的过程了。示例如下：

```
from queue import Queue
from threading import Thread, Event
```

```
# A thread that produces data
def producer(out_q):
    while running:
        # Produce some data
        ...
        # Make an (data, event) pair and hand it to the consumer
        evt = Event()
        out_q.put((data, evt))
        ...
        # Wait for the consumer to process the item
        evt.wait()

# A thread that consumes data
def consumer(in_q):
    while True:
        # Get some data
        data, evt = in_q.get()
        # Process the data
        ...
        # Indicate completion
        evt.set()
```

## 讨论¶

基于简单队列编写多线程程序在多数情况下是一个比较明智的选择。从线程安全队列的底层实现来看，你无需在你的代码中使用锁和其他底层的同步机制，这些只会把你的程序弄得乱七八糟。此外，使用队列这种基于消息的通信机制可以被扩展到更大的应用范畴，比如，你可以把你的程序放入多个进程甚至是分布式系统而无需改变底层的队列结构。 使用线程队列有一个要注意的问题是，向队列中添加数据项时并不会复制此数据项，线程间通信实际上是在线程间传递对象引用。如果你担心对象的共享状态，那你最好只传递不可修改的数据结构（如：整型、字符串或者元组）或者一个对象的深拷贝。例如：

```
from queue import Queue
from threading import Thread
import copy

# A thread that produces data
def producer(out_q):
    while True:
        # Produce some data
        ...
        out_q.put(copy.deepcopy(data))

# A thread that consumes data
def consumer(in_q):
    while True:
        # Get some data
        data = in_q.get()
        # Process the data
        ...
```

`Queue` 对象提供一些在当前上下文很有用的附加特性。比如在创建 Queue 对象时提供可选的 `size` 参数来限制可以添加到队列中的元素数量。对于“生产者”与“消费者”速度有差异的情况，为队列中的元素数量添加上限是有意义的。比如，一个“生产者”产生项目的速度比“消费者”“消费”的速度快，那么使用固定大小的队列就可以在队列已满的时候阻塞队列，以免未预期的连锁效应扩散整个程序造成死锁或者程序运行失常。在通信的线程之间进行“流量控制”是一个看起来容易实现起来困难的问题。如果你发现自己曾经试图通过摆弄队列大小来解决一个问题，这也许就标志着你的程序可能存在脆弱设计或者固有的可伸缩问题。 `get()` 和 `put()` 方法都支持非阻塞方式和设定超时，例如：

```
import queue
q = queue.Queue()

try:
    data = q.get(block=False)
except queue.Empty:
    ...

try:
    q.put(item, block=False)
```

```
except queue.Full:
    ...

try:
    data = q.get(timeout=5.0)
except queue.Empty:
    ...
```

这些操作都可以用来避免当执行某些特定队列操作时发生无限阻塞的情况，比如，一个非阻塞的 `put()` 方法和一个固定大小的队列一起使用，这样当队列已满时就可以执行不同的代码。比如输出一条日志信息并丢弃。

```
def producer(q):
    ...
    try:
        q.put(item, block=False)
    except queue.Full:
        log.warning('queued item %r discarded!', item)
```

如果你试图让消费者线程在执行像 `q.get()` 这样的操作时，超时自动终止以便检查终止标志，你应该使用 `q.get()` 的可选参数 `timeout`，如下：

```
_running = True

def consumer(q):
    while _running:
        try:
            item = q.get(timeout=5.0)
            # Process item
            ...
        except queue.Empty:
            pass
```

最后，有 `q.qsize()`， `q.full()`， `q.empty()` 等实用方法可以获取一个队列的当前大小和状态。但要注意，这些方法都不是线程安全的。可能你对一个队列使用 `empty()` 判断出这个队列为空，但同时另外一个线程可能已经向这个队列中插入一个数据项。所以，你最好不要在你的代码中使用这些方法。