

# 分发

## 生产者消费者模型

对于一个监控系统，需要处理很多数据，包括日志。对其中已有数据的采集、分析。被监控对象就是数据的生产者producer，数据的处理程序就是数据的消费者consumer。

生产者消费者传统模型



传统的生产者消费者模型，生产者生产，消费者消费。但这种模型有些问题开发的代码耦合太高，如果生成规模扩大，不易扩展，生产和消费的速度很难匹配等。

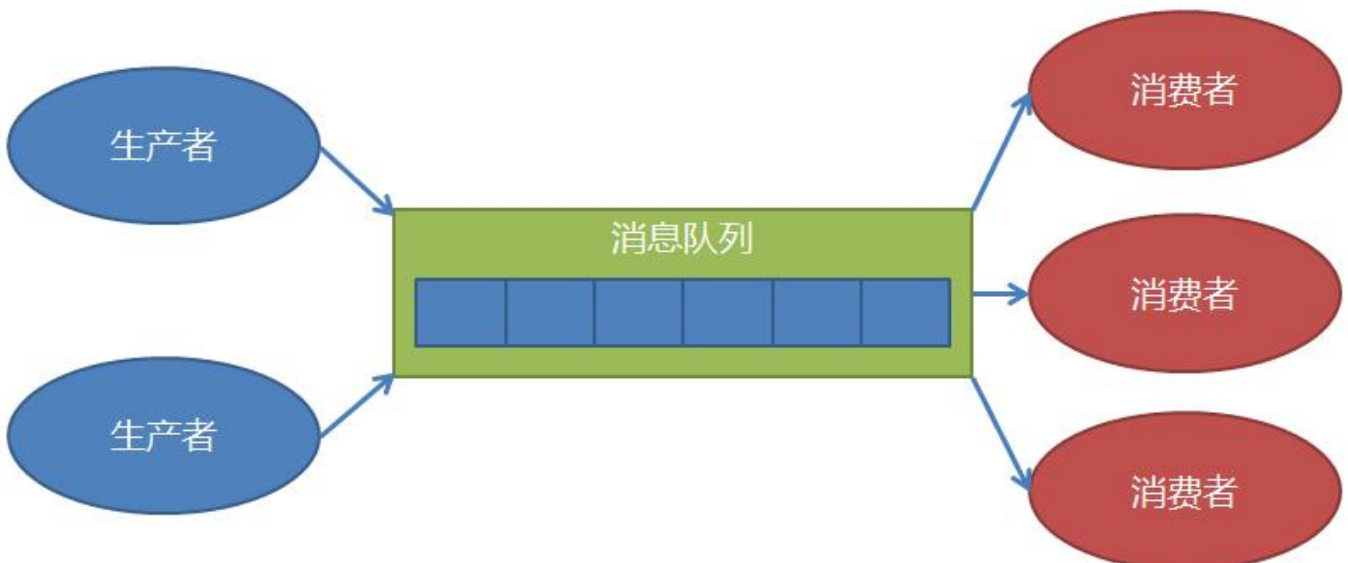
思考一下，生产者和消费者的问题是什么？

举例：

卖包子的，如果包子卖不完，还要继续蒸包子，会怎么样？门可罗雀，包子成山。  
如果把包子先蒸一些，卖着，快卖完了，赶紧包，再蒸一些。不会有等包子的队伍。  
如果包子供不应求，还没有和面呢，包子都被预定了，出现排队等包子的情况。  
上面这些情况，最核心的问题，就是生产者和消费者速度要匹配的问题。  
但是，往往速度不能够很好的匹配。

解决的办法——队列queue。

作用——解耦、缓冲。



日志生产者往往会部署好几个程序，日志产生的也很多，而消费者也会有多个程序，去提取日志

分析处理。

数据的生产是不稳定的！会造成短时间数据的“潮涌”，需要缓冲。

消费者消费能力不一样，有快有慢，消费者可以自己决定消费缓冲区中的数据。

单机可以使用queue内建的模块构建进程内的队列，满足多个线程间的生产消费需要。

大型系统可以使用第三方消息中间件：RabbitMQ、RocketMQ、Kafka

## queue模块——队列

queue模块提供了一个先进先出的队列Queue。

`queue.Queue(maxsize=0)`

创建FIFO队列，返回Queue对象。

maxsize 小于等于0，队列长度没有限制。

`Queue.get(block=True, timeout=None)`

从队列中移除元素并返回这个元素。

block 为阻塞，timeout为超时。

如果block为True，是阻塞，timeout为None就是一直阻塞。

如果block为True但是timeout有值，就阻塞到一定秒数抛出Empty异常。

block为False，是非阻塞，timeout将被忽略，要么成功返回一个元素，要么抛出empty异常。

`Queue.get_nowait()`

等价于 `get(False)`，也就是说要么成功返回一个元素，要么抛出empty异常。

但是queue的这种阻塞效果，需要多线程的时候演示。

`Queue.put(item, block=True, timeout=None)`

把一个元素加入到队列中去。

block=True，timeout=None，一直阻塞直至有空位放元素。

block=True，timeout=5，阻塞5秒就抛出Full异常。

block=False，timeout失效，立即返回，能塞进去就塞，不能则返回抛出Full异常。

`Queue.put_nowait(item)`

等价于 `put(item, False)`，也就是能塞进去就塞，不能则返回抛出Full异常。

```
# Queue测试
from queue import Queue
import random

q = Queue()
```

```
q.put(random.randint(1,100))
q.put(random.randint(1,100))

print(q.get())
print(q.get())
#print(q.get()) # 阻塞
print(q.get(timeout=3)) # 阻塞，但超时抛异常
```

生产者（数据源）生产数据，缓冲到消息队列中

由于一条数据会被多个不同的注册过的**handler**处理，所以最好的方式是多线程。

## 线程使用举例

```
import threading

# 定义线程
# target线程中运行的函数；args这个函数运行时需要实参的元组
t = threading.Thread(target=window, args=(src, handler, width, interval))

# 启动线程
t.start()
```

## 分发器代码实现

```
import random
import datetime
import time
from queue import Queue
import threading

def source(second=1):
    """生成数据"""
    while True:
        yield {
            'datetime': datetime.datetime.now(datetime.timezone(datetime.timedelta(hours=8))),
            'value': random.randint(1, 100)
        }
        time.sleep(second)

def window(src: Queue, handler, width: int, interval: int):
    """
    窗口函数

    :param src: 数据源，缓存队列，用来拿数据
    :param handler: 数据处理函数
    :param width: 时间窗口宽度，秒
    """
```

**:param interval:** 处理时间间隔, 秒

"""

start = datetime.datetime.strptime('20170101 000000 +0800', '%Y%m%d %H%M%S %z')

current = datetime.datetime.strptime('20170101 010000 +0800', '%Y%m%d %H%M%S %z')

)

buffer = [] # 窗口中的待计算数据

delta = datetime.timedelta(seconds=width-interval)

**while True:**

# 从数据源获取数据

data = src.get()

**if** data:

buffer.append(data) # 存入临时缓冲等待计算

current = data['datetime']

# 每隔interval计算buffer中的数据一次

**if** (current - start).total\_seconds() >= interval:

ret = handler(buffer)

print('{:.2f}'.format(ret))

start = current

# 清除超出width的数据

buffer = [x **for** x **in** buffer **if** x['datetime'] > current - delta]

**def** handler(iterable):

**return** sum(map(**lambda** x:x['value'], iterable)) / len(iterable)

**def** dispatcher(src):

# 分发器中记录handler, 同时保存各自的队列

handlers = []

queues = []

**def** reg(handler, width:int, interval:int):

"""

注册 窗口处理函数

**:param handler:** 注册的数据处理函数

**:param width:** 时间窗口宽度

```
:param interval: 时间间隔
```

```
"""
```

```
q = Queue()
```

```
queues.append(q)
```

```
h = threading.Thread(target=window, args=(q, handler, width, interval))
```

```
handlers.append(h)
```

```
def run():
```

```
    for t in handlers:
```

```
        t.start() # 启动线程处理数据
```

```
    for item in src: # 将数据源取到的数据分发到所有队列中
```

```
        for q in queues:
```

```
            q.put(item)
```

```
    return reg, run
```

```
reg, run = dispatcher(source())
```

```
reg(handler, 10, 5) # 注册
```

```
run() # 运行
```

注意，以上代码也只是现阶段所学知识的一种实现，项目中建议使用消息队列服务的“订阅”模式，消费者各自消费自己的队列的数据。