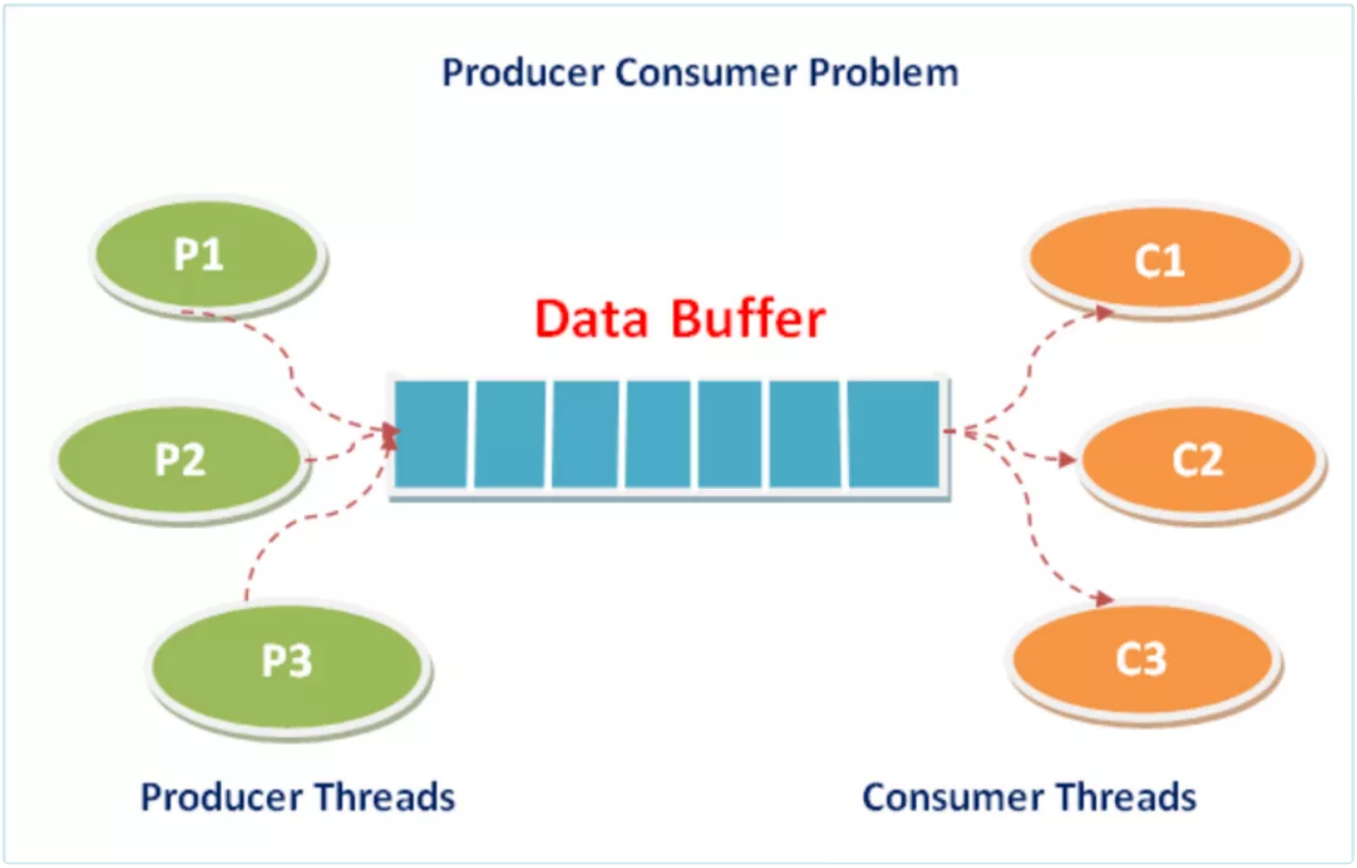
孔乙己：Kotlin 生产者消费者问题的 8 种解法

Original fundroid AndroidPub Yesterday



生产者和消费者问题是线程模型中的经典问题：生产者和消费者在同一时间段内共用同一个缓冲区（Buffer），生产者往 Buffer 中添加产品，消费者从 Buffer 中取走产品，当 Buffer 为空时，消费者阻塞，当 Buffer 满时，生产者阻塞。

Kotlin 中有多种方法可以实现多线程的生产/消费模型（大多也适用于Java）

1. Synchronized
2. ReentrantLock
3. BlockingQueue
4. Semaphore
5. PipedXXXStream
6. RxJava

7. Coroutine
8. Flow

1. Synchronized

Synchronized 是最最基本的线程同步工具，配合 `wait` / `notify` 可以实现实现生产消费问题

```
val buffer = LinkedList<Data>()
val MAX = 5 //buffer最大size

val lock = Object()

fun produce(data: Data) {
    sleep(2000) // mock produce
    synchronized(lock) {
        while (buffer.size >= MAX) {
            // 当buffer满时，停止生产
            // 注意此处使用while不能使用if，因为有可能是被另一个生产线程而非消费线程唤醒，所以要再次检查b
            // 如果生产消费两把锁，则不必担心此问题
            lock.wait()
        }

      buffer.push(data)
        // notify方法只唤醒其中一个线程，选择哪个线程取决于操作系统对多线程管理的实现。
        // notifyAll会唤醒所有等待中线程，哪一个线程将会第一个处理取决于操作系统的实现，但是都有机会处理。
        // 此处使用notify有可能唤醒的是另一个生产线程从而造成死锁，所以必须使用notifyAll
        lock.notifyAll()
    }
}

fun consume() {
    synchronized(lock) {
        while (buffer.isEmpty())
            lock.wait() // 暂停消费
        buffer.removeFirst()
        lock.notifyAll()
```

```
    }
    sleep(2000) // mock consume
}



@Test
fun test() {
    // 同时启动多个生产、消费线程
    repeat(10) {
        Thread { produce(Data()) }.start()
    }
    repeat(10) {
        Thread { consume() }.start()
    }
}
```

2. ReentrantLock

Lock 相对于 Synchronized 好处是当有多个生产线/消费线程时，我们可以通过定义多个 `condition` 精确指定唤醒哪一个。下面的例子展示 Lock 配合 `await` / `single` 替换前面 Synchronized 写法

```
val buffer = LinkedList<Data>()
val MAX = 5 //buffer最大size

val lock = ReentrantLock()
val condition = lock.newCondition()

fun produce(data: Data) {
    sleep(2000) // mock produce
    lock.lock()
```

```
    while (buffer.size >= 5)
        condition.await()

    buffer.push(data)
    condition.signalAll()
    lock.unlock()
}

fun consume() {
    lock.lock()
    while (buffer.isEmpty())
        condition.await()

    buffer.removeFirst()
    condition.singleAll()
    lock.unlock()
    sleep(2000) // mock consume
}
```

3. BlockingQueue (阻塞队列)

BlockingQueue在达到临界条件时，再进行读写会自动阻塞当前线程等待锁的释放，天然适合这种生产/消费场景

```
val buffer = LinkedBlockingQueue<Data>(5)

fun produce(data: Data) {
    sleep(2000) // mock produce
    buffer.put(data) //buffer满时自动阻塞
}

fun consume() {
    buffer.take() // buffer空时自动阻塞
    sleep(2000) // mock consume
```

```
}
```

注意 BlockingQueue 的有三组读/写方法，只有一组有阻塞效果，不要用错

方法	说明
add(o)/remove(o)	add 方法在添加元素的时候，若超出了队列的长度会直接抛出异常
offer(o)/poll(o)	offer 在添加元素时，如果发现队列已满无法添加的话，会直接返回false
put(o)/take(o)	put 向队尾添加元素的时候发现队列已经满了会发生阻塞一直等待空间，以加入元素

4. Semaphore（信号量）

Semaphore 是 JUC 提供的一种共享锁机制，可以进行拥塞控制，此特性可用来控制 buffer 的大小。

```
// canProduce: 可以生产数量（即buffer可用的数量），生产者调用acquire，减少permit数目
val canProduce = Semaphore(5)
// canConsumer：可以消费数量，生产者调用release，增加permit数目
val canConsume = Semaphore(5)
// 控制buffer访问互斥
val mutex = Semaphore(0)

val buffer = LinkedList<Data>()

fun produce(data: Data) {
    if (canProduce.tryAcquire()) {
        sleep(2000) // mock produce

        mutex.acquire()
```

```
        buffer.push(data)
        mutex.release()

        canConsume.release() //通知消费端新增加了一个产品
    }
}

fun consume() {
    if (canConsume.tryAcquire()) {
        sleep(2000) // mock consume

        mutex.acquire()
        buffer.removeFirst()
        mutex.release()

        canProduce.release() //通知生产端可以再追加生产
    }

}
```

5. PipedXXXStream (管道)

Java 里的管道输入/输出流 `PipedInputStream` / `PipedOutputStream` 实现了类似管道的功能，用于不同线程之间的相互通信，输入流中有一个缓冲数组，当缓冲数组为空的时候，输入流 `PipedInputStream` 所在的线程将阻塞

```
val pis: PipedInputStream = PipedInputStream()
val pos: PipedOutputStream by lazy {
    PipedOutputStream().apply {
        pis.connect(this) //输入输出流之间建立连接
    }
}

fun produce(data: ContactsContract.Data) {
    while (true) {
```

```
        sleep(2000)
        pos.use { // Kotlin 使用 use 方便的进行资源释放
            it.write(data.getBytes())
            it.flush()
        }
    }
}

fun consume() {
    while (true) {
        sleep(2000)
        pis.use {
            val byteArray = ByteArray(1024)
            it.read(byteArray)
        }
    }
}

@Test
fun Test() {
    repeat(10) {
        Thread { produce(Data()) }.start()
    }

    repeat(10) {
        Thread { consume() }.start()
    }
}
```

6. RxJava

RxJava 从概念上，可以将 `Observable` / `Subject` 作为生产者， `Subscriber` 作为消费者，但是无论 Subject 或是 Observable 都缺少 Buffer 溢出时的阻塞机制，难以独立实现生产者/消费者模型。

`Flowable` 的背压机制，可以用来控制 buffer 数量，并在上下游之间建立通信， 配合 `Atomic` 可以变向实现单生产者/单消费者场景，（不适用于多生产者/多消费者场景）。

```
class Producer : Flowable<Data>() {

    override fun subscribeActual(subscriber: org.reactivestreams.Subscriber<in Data>) {
        subscriber.onSubscribe(object : Subscription {
            override fun cancel() {
                //...
            }

            private val outStandingRequests = AtomicLong(0)

            override fun request(n: Long) { //收到下游通知，开始生产
                outStandingRequests.addAndGet(n)

                while (outStandingRequests.get() > 0) {
                    sleep(2000)
                    subscriber.onNext(Data())
                    outStandingRequests.decrementAndGet()
                }
            }

        })
    }

}



class Consumer : DefaultSubscriber<Data>() {

    override fun onStart() {
        request(1)
    }

    override fun onNext(i: Data?) {
        sleep(2000) //mock consume
        request(1) //通知上游可以增加生产
    }

    override fun onError(throwable: Throwable) {
```

```
        //...
    }

    override fun onComplete() {
        //...
    }

}


@Test
fun test_rxjava() {
    try {
        val testProducer = Producer)
        val testConsumer = Consumer()

        testProducer
            .subscribeOn(Schedulers.computation())
            .observeOn(Schedulers.single())
            .blockingSubscribe(testConsumer)

    } catch (t: Throwable) {
        t.printStackTrace()
    }

}
```

7. Coroutine Channel

协程中的 `Channel` 具有拥塞控制机制，可以实现生产者消费者之间的通信。可以把 `Channel` 理解为一个协程版本的阻塞队列， `capacity` 指定队列容量。

```
val channel = Channel<Data>(capacity = 5)

suspend fun produce(data: ContactsContract.Contacts.Data) = run {
```

```
suspend fun produce(data: ContactsContract.Contacts.Data) = run {
    delay(2000) //mock produce
    channel.send(data)
}


suspend fun consume() = run {
    delay(2000)//mock consume
    channel.receive()
}

@Test
fun test_channel() {
    repeat(10) {
        GlobalScope.launch {
            produce(Data())
        }
    }

    repeat(10) {
        GlobalScope.launch {
           consume()
        }
    }
}
```

此外，Coroutine 提供了 `produce` 方法，在声明 Channel 的同时生产数据，写法上更简单，适合单消费者单生产者的场景：

```
fun CoroutineScope.produce(): ReceiveChannel<Data> = produce {
    repeat(10) {
        delay(2000) //mock produce
        send(Data())
    }
}

@Test
fun test_produce() {
    GlobalScope.launch {
        produce.consumeEach {
            delay(2000) //mock consume
```

```
        }
    }
}
```

8. Coroutine Flow

Flow 跟 RxJava 一样，因为缺少 Buffer 溢出时的阻塞机制，不适合处理生产消费问题，其背压机制也比较简单，无法像 RxJava 那样收到下游通知。但是 Flow 后来发布了 `SharedFlow` ， 作为带缓冲的热流，提供了 Buffer 溢出策略，可以用作生产者/消费者之间的同步。

```
val flow : MutableSharedFlow<Data> = MutableSharedFlow(
    extraBufferCapacity = 5  //缓冲大小
    , onBufferOverflow = BufferOverflow.SUSPEND // 缓冲溢出时的策略：挂起
)

@Test
fun test() {

    GlobalScope.launch {
        repeat(10) {
            delay(2000) //mock produce
            sharedFlow.emit(Data())
        }
    }

    GlobalScope.launch {
        sharedFlow.collect {
            delay(2000) //mock consume
        }
    }
}
```

注意 `SharedFlow` 也只能用在单生产者/单消费者场景

总结

生产者/消费者问题，其本质核心还是多线程读写共享资源（Buffer）时的同步问题，理论上只要具有同步机制的多线程框架，例如线程锁、信号量、阻塞队列、协程 Channel等，都是可以实现生产消费模型的。

另外，RxJava 和 Flow 虽然也是多线程框架，但是缺少Buffer溢出时的阻塞机制，不适用于生产/消费场景，更适合在纯响应式场景中使用。

END

推荐阅读

- 100 行写一个 Compose 版华容道
- 建议收藏！Kotlin 线程同步的 N 种方法
- 大厂卡学历？双非二本字节实习面经

↓关注公众号↓	↓添加微信交流↓
	

Read more

People who liked this content also liked

Bootstrap 代码

暮暮学编程

7.Golang作用域详细学习

编程第一天

无涯教程:Node.js - IO介绍

Learnfk无涯教程网