发件人：方堃 fangkun119@icloud.com
主题：CH07 Producer-Consumer
日期：2017年5月2日 下午12:00
收件人：

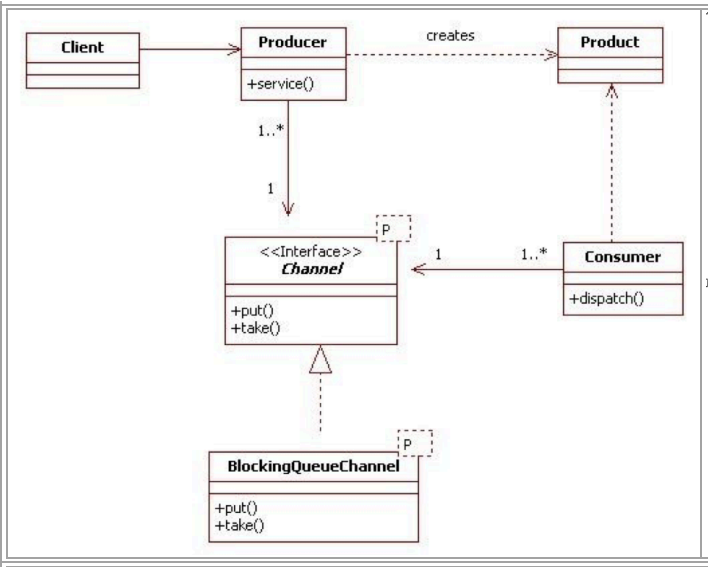
CH07 Producer-Consumer

2017年4月26日 星期三
下午7:59

内容

数据的提供方可形象地称为数据的生产者，它“生产”了数据，而数据的加工方则相应地被称为消费者，它“消费”了数据。实际上，生产者“生产”数据的速率和消费者“消费”数据的速率往往是不均衡的，比如数据的“生产”要比其“消费”快。为了避免数据的生产者和消费者中处理速率快的一方需要等待处理速率慢的一方，Producer-Consumer模式通过在数据的生产者和消费者之间引入一个通道（Channel，暂时可以将其简单地理解为一个队列）对二者进行解耦（Decoupling）：生产者将其“生产”的数据放入通道，消费者从相应通道中取出数据进行“消费”（处理），生产者和消费者各自运行在各自的线程中，从而使双方处理速率互不影响。

类图



- **Producer**：生产者，负责生成相应的“产品”并将其存入通道。其主要方法及职责如下。
  - **service**：生产者对外暴露的服务方法。
- **Product**：生产者所“生产”的数据或者任务。其具体类型由应用程序决定。
- **Channel**：对通道的抽象。通道充当生产者和消费者之间的缓冲区用于“产品”传递，它可以是有存储容量限制的。该接口的类型参数P相当于表示“产品”的类型。该接口的主要方法及职责如下。
  - **put**：将“产品”存入通道。
  - **take**：从通道中取出一个“产品”。
- **BlockingQueueChannel**：基于阻塞队列（Blocking Queue）的Channel实现。其主要方法及职责如下。
  - **put**：将“产品”存入通道。当队列满时，该方法会将当前线程挂起直到队列非满。
  - **take**：从通道中取出一个“产品”。当队列为空时，该方法通常会将当前线程挂起直到队列非空。
- **Consumer**：消费者，负责对“产品”进行处理。其主要方法及职责如下。
  - **dispatch**：从通道中获取“产品”，并对其进行处理。

时序

生产（假定生产者队列无限制）

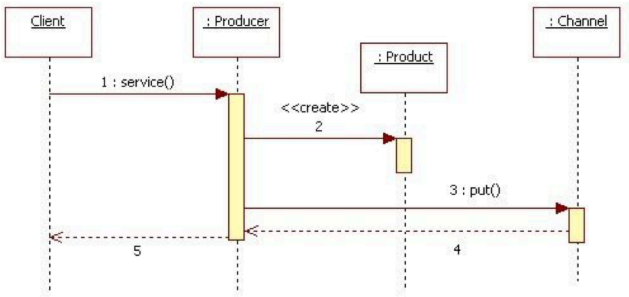


图7-2. Producer-Consumer模式的“产品”“生产”序列图

消费者

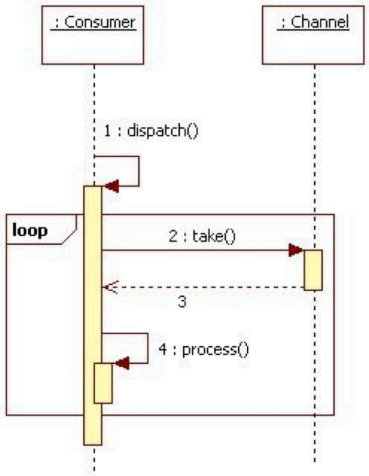


图7-3. Producer-Consumer模式的“产品”“消费”序列图

使用者

Producer-Consumer模式使得“产品”的生产者和消费者各自的处理能力（速率）相对来说互不影响。生产者只需要将其“生产”的“产品”放入通道中就可以继续处理，而不必等待相应的“产品”被消费者处理完毕。而消费者运行在其自身的工作者线程中，它只管从通道中取“产品”进行处理而不必关心这些“产品”由谁“生产”以及如何“生产”这些细节。因而消费者的处理能力相对来说又不影响生产者，同时又与生产者是松耦合（Loose Coupling）的关系。另一方面，当消费者处理能力比生产者处理能力大的时候，可能出现通道为空的情形，此时消费者的工作者线程会被暂挂直到生产者“生产”了新的“产品”。此时出现了事实上的消费者等待生产者的情形。类似地，当消费者的处理能力小于生产者的处理能力时，通道可能会满，导致生产者线程被暂挂直到消费者“消费”了通道中的部分“产品”而腾出了存储空间。此时出现了事实上的生产者等待消费者的情形。因此，我们说生产者和消费者各自的处理能力相互不影响是相对的。

通道积压问题

问题	生产者生产速度大于消费者时，通道会发生挤压，某些场景需要控制生产者的生产速率，有两种方法
方法	• 使用有界阻塞队列。使用有界阻塞队列（如ArrayBlockingQueue和带容量限制的LinkedBlockingQueue）作为Channel参与者的实现可以实现将消费者处理压力“反弹”给生产者的效果，从而使消费者处理负荷过大时相应的生产者的处理能力也下降一定程度以达到平衡二者处理能力的目的。当消费者处理能力低于生产者的处理能力时，作为通道的有界阻塞队列会逐渐积压到队列满，此时生产者线程会被阻塞直到相应的消费者“消费”了队列中的一些“产品”使得队列非满。也就是出现了生产者的步伐等待消费者的步伐的情形。 • 使用带流量控制的无界阻塞队列。使用无界阻塞队列（如不带容量限制LinkedBlockingQueue）作为Channel参与者的实现也可以实现平衡生产者和消费者的处理能力。这通常是借助流量控制实现的，即对同一时间内可以有多少个生产者线程往通道中存储“产品”进行限制，从而达到平衡生产者和消费者的处理能力的目的，如清单7-4所示。
方法1	复用JDK提供的带容量限制的BlockingQueue就可以实现

方法2 用semaphore来控制同时执行put()操作的线程数量

```
public class SemaphoreBasedChannel<P> implements Channel<P> {
 private final BlockingQueue<P> queue;
 private final Semaphore semaphore;

 /**
  *
  * @param queue 阻塞队列，通常是一个无界阻塞队列。
  * @param flowLimit 流量限制数
  */
 public SemaphoreBasedChannel(BlockingQueue<P> queue, int flo
  this.queue = queue;
  this.semaphore = new Semaphore(flowLimit);
 }

 @Override
 public P take() throws InterruptedException {
  return queue.take();
 }

 @Override
 public void put(P product) throws InterruptedException {
  semaphore.acquire();
  try {
   queue.put(product);
  } finally {
   semaphore.release();
  }
 }
}
```

消费者竞争问题

问题	多个消费者同时从同一个阻塞队列中取数据时，会引发互斥（获取队头元素）
解决	step1：每个消费者都有自己的队列，减少阻塞 step2：一个消费者的队列消费完之后，可以访问其他消费者的队列（消费者窃取算法），避免消费者队列不均

Producer-Consumer模式中的通道通常可以使用队列来实现。一个通道可以对应一个或者多个队列实例。本章案例中（代码见清单7-1），一个通道仅对应一个队列（ArrayBlockingQueue）实例。这意味着，如果有多个消费者从该通道中获取“产品”，那么这些消费者的工作者线程实际上是在共享同一个队列实例，而这会导致锁的竞争，即修改队列的头指针时所需要获得的锁而导致的竞争。如果一个通道实例对应多个队列实例，那么就可以实现多个消费者线程从通道中取“产品”的时候访问的是各自的队列实例。此时，各个消费者线程修改队列的头指针并不会导致锁竞争。

一个通道实例对应多个队列实例的时候，当一个消费者线程处理完该线程对应的队列中的“产品”时，它可以继续从其他消费者线程对应的队列中取出“产品”进行处理，这样就不会导致该消费者线程闲置，并减轻其他消费者线程的负担。这就是工作窃取（Work Stealing）算法的思想。清单7-5展示了一个工作窃取算法的示例代码。

例子	见例子2，消费者窃取算法 + 两阶段终止

高可靠实现 本章中我们给出的Producer-Consumer模式实现可以说是一个比较一般的实现。如果应用程序对准备采用Producer-Consumer模式实现的服务有较高的性能和可靠性的要求，那么不妨考虑使用开源的Producer-Consumer模式实现库LMAX Disruptor。详情请参阅本章相应的参考资源。

可复用代码1 JDK 1.5引入的标准库类java.util.concurrent.ThreadPoolExecutor可以看成是Producer-Consumer模式的可复用实现。ThreadPoolExecutor内部维护的工作队列和工作者线程相当于Producer-Consumer模式的Channel参与者和Consumer参与者。而ThreadPoolExecutor的客户端代码则相当于Producer参与者。利用ThreadPoolExecutor实现Producer-Consumer模式，应用代码只需要完成以下几件事情。

1. 【必需】创建Runnable实例（任务），该实例相当于“产品”。
2. 【必需】客户端代码调用ThreadPoolExecutor实例的submit方法提交一个任务，这相当于Producer往通道中放入一个“产品”。

有关ThreadPoolExecutor的进一步信息，可参考Thread Pool模式（第9章）。

可复用代码2

Channel<P>

```
public interface Channel<P> {
 /**
  * 从通道中取出一个“产品”。
  *
  * @return “产品”
  * @throws InterruptedException
  */
 P take() throws InterruptedException;

 /**
  * 往通道中存储一个“产品”。
  *
  * @param product
  *            “产品”
  * @throws InterruptedException
  */
 void put(P product) throws InterruptedException;
}
```

BlockingQueueChannel<P>

```
public class BlockingQueueChannel<P> implements Channel<P> {
 private final BlockingQueue<P> queue;

 public BlockingQueueChannel(BlockingQueue<P> queue) {
  this.queue = queue;
 }

 @Override
 public P take() throws InterruptedException {

  return queue.take();
 }

 @Override
 public void put(P product) throws InterruptedException {
  queue.put(product);

 }

}
```

JDK例 Java标准库中的类java.io.PipedOutStream和java.io.PipedInputStream允许一个线程以I/O的形式输出数据给另外一个线程。这里，java.io.PipedOutStream、java.io.PipedInputStream分别相当于Producer-Consumer模式的Producer参与者和Consumer参与者。而java.io.PipedOutStream内部维护的缓冲区则相当于Producer-Consumer模式的Channel参与者。

例子1 某内容管理系统需要支持对文档附件中的文件（格式包括Word、PDF）进行全文检索（Full-text Search）。该系统中，附件会被上传到专用的文件服务器上，对附件进行全文检索的功能模块也是部署在文件服务器上的。因此，与一份文档相关联的附件被上传到文件服务器之后，我们还需要对这些附件生成相应的索引文件以供后面对附件进行全文检索时使用。对附件生成索引的过程包括文件I/O（读取附件文件和写索引文件）和一些计算（如进行分词），该过程相对于将上传的附件保存到磁盘中而言也快不到哪里。因此，我们不希望对附件生成索引文件这个操作的快慢影响系统用户的体验（如增加了用户等待系统给出操作反馈的时间）。此时，Producer-Consumer模式可以排上用场：我们可以把负责附件存储的线程看作生产者，其“产品”是一个已经保存到磁盘的文件。另外，我们引入一个负责对已存储的附件文件生成相应索引文件的线程，该线程就相当于消费者，它“消费”了上传到文件服务器的附件文件。

该案例的代码如清单7-1所示。其中，负责对上传的附件进行存储的类AttachmentProcessor，它相当于Producer-Consumer模式中的Producer参与者，负责对附件文件生成索引文件的线程indexingThread则相当于Producer-Consumer模式中的Consumer参与者。AttachmentProcessor的实例变量channel相当于Channel参与者实例。AttachmentProcessor将上传的附件保存完毕后，就将相应的文件存入通道channel，便返回了，它不会等待该文件相应的索引文件的生成，因此减少了系统用户的等待时间。而相应文件对应的索引文件由Consumer的工作者线程indexingThread负责生成。工作者线程indexingThread使用了Two-phase Termination模式（参见第5章）以实现该线程的优雅停止。

Channel及BlockingQueueChannel见可复用代码

```
//模式角色：Producer-Consumer.Producer
public class AttachmentProcessor {
 private final String ATTACHMENT_STORE_BASE_DIR =
"/home/viscent/tmp/attachments/";

 // 模式角色：Producer-Consumer.Channel
 private final Channel<File> channel = new BlockingQueueChannel<File>(
     new ArrayBlockingQueue<File>(200));

 // 模式角色：Producer-Consumer.Consumer
 private final AbstractTerminatableThread indexingThread = new
AbstractTerminatableThread() {

  @Override
  protected void doRun() throws Exception {
   File file = null;

   file = channel.take();
   try {
```

```
    indexFile(file);
   } catch (Exception e) {
    e.printStackTrace();
   } finally {
    terminationToken.reservations.decrementAndGet();
   }

  }

  // 根据指定文件生成全文搜索所需的索引文件
  private void indexFile(File file) throws Exception {
   // 省略其他代码

   // 模拟生成索引文件的时间消耗
   Random rnd = new Random();
//模式角色: Producer-Consumer.Producer
public class AttachmentProcessor {
 private final String ATTACHMENT_STORE_BASE_DIR =
"/home/viscent/tmp/attachments/";

 // 模式角色: Producer-Consumer.Channel
 private final Channel<File> channel = new BlockingQueueChannel<File>(
     new ArrayBlockingQueue<File>(200));

 // 模式角色: Producer-Consumer.Consumer
 private final AbstractTerminatableThread indexingThread = new
AbstractTerminatableThread() {

  @Override
  protected void doRun() throws Exception {
   File file = null;

   file = channel.take();
   try {
    indexFile(file);
   } catch (Exception e) {
    e.printStackTrace();
   } finally {
    terminationToken.reservations.decrementAndGet();
   }

  }

  // 根据指定文件生成全文搜索所需的索引文件
  private void indexFile(File file) throws Exception {
   // 省略其他代码

   // 模拟生成索引文件的时间消耗
   Random rnd = new Random();
   try {
    Thread.sleep(rnd.nextInt(100));
   } catch (InterruptedException e) {
    ;
   }
  }

 };

 public void init() {
  indexingThread.start();
 }

 public void shutdown() {
  indexingThread.terminate();
 }

 public void saveAttachment(InputStream in, String documentId,
     String originalFileName) throws IOException {
  File file = saveAsFile(in, documentId, originalFileName);
  try {
   channel.put(file);
  } catch (InterruptedException e) {
   ;
  }
  indexingThread.terminationToken.reservations.incrementAndGet();
 }

 private File saveAsFile(InputStream in, String documentId,
     String originalFileName) throws IOException {
  String dirName = ATTACHMENT_STORE_BASE_DIR + documentId;
  File dir = new File(dirName);
  dir.mkdirs();
  File file = new File(dirName + '/'
      + Normalizer.normalize(originalFileName, Normalizer.Form.NFC));
  // 防止目录跨越攻击
  if (! dirName.equals(file.getCanonicalFile(). getParent())) {
   throw new SecurityException("Invalid originalFileName:"
       + originalFileName);
  }

  BufferedOutputStream bos = null;
  BufferedInputStream bis = new BufferedInputStream(in);
  byte[] buf = new byte[2048];
  int len = -1;
  try {
   bos = new BufferedOutputStream(new FileOutputStream(file));
   while ((len = bis.read(buf)) > 0) {
    bos.write(buf, 0, len);
   }
   bos.flush();
  } finally {
   try {
    bis.close();
   } catch (IOException e) {
    .
```

```
      ,
     }
     try {
      if (null != bos) {
       bos.close();
      }
     } catch (IOException e) {
      ;
     }
    }

    return file;
   }

}
```

例子2

内容	消费者窃取算法 + 两阶段终止

解决问题

Producer-Consumer模式中的通道通常可以使用队列来实现。一个通道可以对应一个或者多个队列实例。本章案例中（代码见清单7-1），一个通道仅对应一个队列（ArrayBlockingQueue）实例。这意味着，如果有多个消费者从该通道中获取“产品”，那么这些消费者的工作者线程实际上是在共享同一个队列实例，而这会导致锁的竞争，即修改队列的头指针时所需要获得的锁而导致的竞争。如果一个通道实例对应多个队列实例，那么就可以实现多个消费者线程从通道中取“产品”的时候访问的是各自的队列实例。此时，各个消费者线程修改队列的头指针并不会导致锁竞争。

一个通道实例对应多个队列实例的时候，当一个消费者线程处理完该线程对应的队列中的“产品”时，它可以继续从其他消费者线程对应的队列中取出“产品”进行处理，这样就不会导致该消费者线程闲置，并减轻其他消费者线程的负担。这就是工作窃取（Work Stealing）算法的思想。清单7-5展示了一个工作窃取算法的示例代码。

代码

```
/**
 * 工作窃取算法示例。该类使用Two-phase Termination模式（参见第5章）。
 *
 * @author Viscent Huang
 *
 */
public class WorkStealingExample {
 private final WorkStealingEnabledChannel<String> channel;
 private final TerminationToken token = new TerminationToken();

 public WorkStealingExample()  {
  int nCPU = Runtime.getRuntime(). availableProcessors();
  int consumerCount = nCPU / 2 + 1;

  @SuppressWarnings("unchecked")
  BlockingDeque<String>[]  managedQueues = new LinkedBlockingDeque[consumerCount];

  // 该通道实例对应了多个队列实例managedQueues
  channel = new WorkStealingChannel<String>(managedQueues);

  Consumer[]  consumers = new Consumer[consumerCount];
  for (int i = 0; i < consumerCount; i++) {
   managedQueues[i] = new LinkedBlockingDeque<String>();
   consumers[i] = new Consumer(token, managedQueues[i]);

  }

  for (int i = 0; i < nCPU; i++) {
   new Producer(). start();
  }

  for (int i = 0; i < consumerCount; i++) {
   consumers[i]. start();
  }

 }

 public void doSomething()  {

 }

 public static void main(String[]  args) throws InterruptedException {
  WorkStealingExample wse;
  wse = new WorkStealingExample();

  wse.doSomething();
  Thread.sleep(3500);

 }
 private class Producer extends AbstractTerminatableThread {
  private int i = 0;

  @Override
  protected void doRun()  throws Exception {
   channel.put(String.valueOf(i++));
   token.reservations.incrementAndGet();
  }

 }

 private class Consumer extends AbstractTerminatableThread {
  private final BlockingDeque<String> workQueue;

  public Consumer(TerminationToken token, BlockingDeque<String> workQueue) {
   super(token);
   this.workQueue = workQueue;
  }

  @Override
  protected void doRun()  throws Exception {

   /*
    * WorkStealingEnabledChannel接口的take(BlockingDequepreferedQueue)方法
    * 实现了工作窃取算法
    */
   String product = channel.take(workQueue);

   System.out.println("Processing product:" + product);

   // 模拟执行真正操作的时间消耗
   try {
    Thread.sleep(new Random(). nextInt(50));
   } catch (InterruptedException e) {
```

```
    }
   } finally {
    token.reservations.decrementAndGet();
   }

  }

 }
}
```

```
public interface WorkStealingEnabledChannel<P> extends Channel<P> {
 P take(BlockingDeque<P> preferredQueue) throws InterruptedException;
}
```

清单7-7. 类WorkStealingChannel的源码

```
public class WorkStealingChannel<T> implements WorkStealingEnabledChannel<T> {
 // 受管队列
 private final BlockingDeque<T>[]  managedQueues;

 public WorkStealingChannel(BlockingDeque<T>[]  managedQueues) {
  this.managedQueues = managedQueues;
 }

 @Override
 public T take(BlockingDeque<T> preferredQueue) throws InterruptedException {

  // 优先从指定的受管队列中取“产品”
  BlockingDeque<T> targetQueue = preferredQueue;
  T product = null;

  // 试图从指定的队列队首取“产品”
  if (null != targetQueue) {
   product = targetQueue.poll();
  }

  int queueIndex = -1;

  while (null == product) {
   queueIndex = (queueIndex + 1) % managedQueues.length;
   targetQueue = managedQueues[queueIndex];
   // 试图从其他受管队列的队尾“窃取”产品”
   product = targetQueue.pollLast();
   if (preferredQueue == targetQueue) {
    break;
   }
  }
```

```
  if (null == product) {

   // 随机“窃取”其他受管队列的“产品”
   queueIndex = (int) (System.currentTimeMillis()  % managedQueues.length);
   targetQueue = managedQueues[queueIndex];
   product = targetQueue.takeLast();
   System.out.println("stealed from " + queueIndex + ":" + product);
  }

  return product;
 }

 @Override
 public void put(T product) throws InterruptedException {
  int targetIndex = (product.hashCode()  % managedQueues.length);
  BlockingQueue<T> targetQueue = managedQueues[targetIndex];
  targetQueue.put(product);
 }

 @Override
 public T take()  throws InterruptedException {
  return take(null);
 }

}
```