15 | Lock和Condition（下）：Dubbo如何用管程实现异步转同步?

王宝令 2019-04-02

00:00 09:05

讲述：王宝令 大小：8.33M

在上一篇文章中，我们讲到 Java SDK 并发包里的 Lock 有别于 synchronized 隐式锁的三个特性：能够响应中断、支持超时和非阻塞地获取锁。那今天我们接着再来详细聊聊 Java SDK 并发包里的 Condition，**Condition 实现了管程模型里面的条件变量**。

在《08 | 管程：并发编程的万能钥匙》里我们提到过 Java 语言内置的管程里只有一个条件变量，而 Lock&Condition 实现的管程是支持多个条件变量的，这是二者的一个重要区别。

在很多并发场景下，支持多个条件变量能够让我们的并发程序可读性更好，实现起来也更容易。例如，实现一个阻塞队列，就需要两个条件变量。

那如何利用两个条件变量快速实现阻塞队列呢?

一个阻塞队列，需要两个条件变量，一个是队列不空（空队列不允许出队），另一个是队列不满（队列已满不允许入队），这个例子我们前面在介绍管程的时候详细说过，这里就不再赘述。相关的代码，我这里重新列了出来，你可以温故知新一下。

复制代码

```
public class BlockedQueue<T>{
  final Lock lock =
    new ReentrantLock();
  // 条件变量：队列不满
  final Condition notFull =
    lock.newCondition();
  // 条件变量：队列不空
  final Condition notEmpty =
    lock.newCondition();

  // 入队
  void enq(T x) {
    lock.lock();
    try {
      while (队列已满){
        // 等待队列不满
        notFull.await();
      }
      // 省略入队操作...
      // 入队后，通知可出队
      notEmpty.signal();
    }finally {
      lock.unlock();
    }
  }
  // 出队
  void deq(){
    lock.lock();
    try {
      while (队列已空){
        // 等待队列不空
        notEmpty.await();
      }
      // 省略出队操作...
      // 出队后，通知可入队
      notFull.signal();
    }finally {
      lock.unlock();
    }
  }
}

```

不过，这里你需要注意，Lock 和 Condition 实现的管程，**线程等待和通知需要调用 await()、signal()、signalAll()**，它们的语义和 wait()、notify()、notifyAll() 是相同的。但是不一样的是，Lock&Condition 实现的管程里只能使用前面的 await()、signal()、signalAll()，而后面的 wait()、notify()、notifyAll() 只有在 synchronized 实现的管程里才能使用。如果一不小心在 Lock&Condition 实现的管程里调用了 wait()、notify()、notifyAll()，那程序可就彻底玩儿完了。

Java SDK 并发包里的 Lock 和 Condition 不过就是管程的一种实现而已，管程你已经很熟悉了，那 Lock 和 Condition 的使用自然是小菜一碟。下面我们就来看看在知名项目 Dubbo 中，Lock 和 Condition 是怎么用的。不过在开始介绍源码之前，我还先要介绍两个概念：同步和异步。

同步与异步

我们平时写的代码，基本都是同步的。但最近几年，异步编程大火，那同步和异步的区别到底是什么呢？**通俗点来讲就是调用方是否需要等待结果，如果调用方不需要等待结果，就是异步。**

比如在下面的代码里，有一个计算圆周率小数点后 100 万位的方法`pai1M()`，这个方法可能需要执行俩礼拜，如果调用`pai1M()`之后，线程一直等着计算结果，等俩礼拜之后结果返回，就可以执行 `printf("hello world")`了，这个属于同步；如果调用`pai1M()`之后，线程不用等待计算结果，立刻就可以执行 `printf("hello world")`，这个就属于异步。

复制代码

```
// 计算圆周率小说点后 100 万位
String pai1M() {
  // 省略代码无数
}

pai1M()
printf("hello world")

```

同步，是 Java 代码默认的处理方式。如果你想让你的程序支持异步，可以通过下面两种方式来实现：

1. 调用方创建一个子线程，在子线程中执行方法调用，这种调用我们称为异步调用；
2. 方法实现的时候，创建一个新的线程执行主要逻辑，主线程直接 return，这种方法我们一般称为异步方法。

Dubbo 源码分析

其实在编程领域，异步的场景还是挺多的，比如 TCP 协议本身就是异步的，我们工作中经常用到的 RPC 调用，**在 TCP 协议层面，发送完 RPC 请求后，线程是不会等待 RPC 的响应结果的**。可能你会觉得奇怪，平时工作中的 RPC 调用大多数都是同步的啊？这是怎么回事呢？

其实很简单，一定是有人帮你做了异步转同步的事情。例如目前知名的 RPC 框架 Dubbo 就给我们做了异步转同步的事情，那它是怎么做的呢？下面我们就来分析一下 Dubbo 的相关源码。

对于下面一个简单的 RPC 调用，默认情况下 sayHello() 方法，是个同步方法，也就是说，执行 service.sayHello（“dubbo”）的时候，线程会停下来等结果。

复制代码

```
DemoService service = 初始化部分省略
String message =
  service.sayHello("dubbo");
System.out.println(message);

```

如果此时你将调用线程 dump 出来的话，会是下图这个样子，你会发现调用线程阻塞了，线程状态是 TIMED_WAITING。本来发送请求是异步的，做了异步转同步的事情。通过调用栈，你能看到线程是阻塞在 DefaultFuture.get() 方法上，所以可以推断：Dubbo 异步转同步的功能应该是通过 DefaultFuture 这个类实现的。

```
"main" #1 prio=5 os_prio=0 tid=0x0000000002c64000 nid=0x1a9c waiting on condition [0x0000000002a7e000]
   java.lang.Thread.State: TIMED_WAITING (parking)
        at sun.misc.Unsafe.park(Native Method)
        - parking to wait for  <0x000000077082a258> (a java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject)
        at java.util.concurrent.locks.LockSupport.parkNanos(LockSupport.java:215)
        at java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.await(AbstractQueuedSynchronizer.java:2163)
        at org.apache.dubbo.remoting.exchange.support.DefaultFuture.get(DefaultFuture.java:177)
        at org.apache.dubbo.remoting.exchange.support.DefaultFuture.get(DefaultFuture.java:164)
        at org.apache.dubbo.rpc.protocol.dubbo.DubboInvoker.doInvoke(DubboInvoker.java:108)
        at org.apache.dubbo.rpc.protocol.AbstractInvoker.invoke(AbstractInvoker.java:157)
        at org.apache.dubbo.rpc.listener.ListenerInvokerWrapper.invoke(ListenerInvokerWrapper.java:78)
        at org.apache.dubbo.monitor.support.MonitorFilter.invoke(MonitorFilter.java:88)
        at org.apache.dubbo.rpc.protocol.ProtocolFilterWrapper$1.invoke(ProtocolFilterWrapper.java:73)
        at org.apache.dubbo.rpc.protocol.dubbo.filter.FutureFilter.invoke(FutureFilter.java:49)
        at org.apache.dubbo.rpc.protocol.ProtocolFilterWrapper$1.invoke(ProtocolFilterWrapper.java:73)
        at org.apache.dubbo.rpc.filter.ConsumerContextFilter.invoke(ConsumerContextFilter.java:54)
        at org.apache.dubbo.rpc.protocol.ProtocolFilterWrapper$1.invoke(ProtocolFilterWrapper.java:73)
        at org.apache.dubbo.rpc.proxy.InvokerInvocationHandler.invoke(InvokerInvocationHandler.java:57)
        at org.apache.dubbo.common.bytecode.proxy0.sayHello(proxy0.java)
        at org.apache.dubbo.demo.consumer.Application.main(Application.java:41)
```

调用栈信息

不过为了理清前后关系，还是有必要分析一下调用 DefaultFuture.get() 之前发生了什么。DubboInvoker 的 108 行调用了 DefaultFuture.get()，这一行很关键，我稍微修改了一下列在了下面。这一行先调用了 request(inv, timeout) 方法，这个方法其实就是发送 RPC 请求，之后通过调用 get() 方法等待 RPC 返回结果。

复制代码

```
public class DubboInvoker{
  Result doInvoke(Invocation inv){
    // 下面这行就是源码中 108 行
    // 为了便于展示，做了修改
    return currentClient
      .request(inv, timeout)
      .get();
  }
}

```

DefaultFuture 这个类是很关键，我把相关的代码精简之后，列到了下面。不过在看代码之前，你还是有必要重复一下我们的需求：当 RPC 返回结果之前，阻塞调用线程，让调用线程等待；当 RPC 返回结果后，唤醒调用线程，让调用线程重新执行。不知道你有没有似曾相识的感觉，这不就是经典的等待 - 通知机制吗？这个时候想必你的脑海里应该能够浮现出管程的解决方案了。有了自己的方案之后，我们再来看看 Dubbo 是怎么实现的。

复制代码

```
// 创建锁与条件变量
private final Lock lock
    = new ReentrantLock();
```

```
4  private final Condition done
5      = lock.newCondition();
6
7  // 调用方通过该方法等待结果
8  Object get(int timeout){
9    long start = System.nanoTime();
10   lock.lock();
11   try {
12         while (!isDone()) {
13           done.await(timeout);
14       long cur=System.nanoTime();
15           if (isDone() ||
16           cur-start > timeout){
17             break;
18           }
19         }
20   } finally {
21         lock.unlock();
22   }
23   if (!isDone()) {
24         throw new TimeoutException();
25   }
26   return returnFromResponse();
27 }
28 // RPC 结果是否已经返回
29 boolean isDone() {
30   return response != null;
31 }
32 // RPC 结果返回时调用该方法
33 private void doReceived(Response res) {
34   lock.lock();
35   try {
36     response = res;
37     if (done != null) {
38       done.signal();
39     }
40   } finally {
41     lock.unlock();
42   }
43 }
44
```

调用线程通过调用 get() 方法等待 RPC 返回结果，这个方法里面，你看到的都是熟悉的“面孔”：调用 lock() 获取锁，在 finally 里面调用 unlock() 释放锁；获取锁后，通过经典的在循环中调用 await() 方法来实现等待。

当 RPC 结果返回时，会调用 doReceived() 方法，这个方法里面，调用 lock() 获取锁，在 finally 里面调用 unlock() 释放锁，获取锁后通过调用 signal() 来通知调用线程，结果已经返回，不用继续等待了。

至此，Dubbo 里面的异步转同步的源码就分析完了，有没有觉得还挺简单的？最近这几年，工作中需要异步处理的越来越多了，其中有一个主要原因就是有些 API 本身就是异步 API。例如 websocket 也是一个异步的通信协议，如果基于这个协议实现一个简单的 RPC，你也会遇到异步转同步的问题。现在很多公有云的 API 本身也是异步的，例如创建云主机，就是一个异步的 API，调用虽然成功了，但是云主机并没有创建成功，你需要调用另外一个 API 去轮询云主机的

状态。如果你需要在项目内部封装创建云主机的 API，你也会面临异步转同步的问题，因为同步的 API 更易用。

总结

Lock&Condition 是管程的一种实现，所以能否用好 Lock 和 Condition 要看你对管程模型理解得怎么样。管程的技术前面我们已经专门用了一篇文章做了介绍，你可以结合着来学，理论联系实践，有助于加深理解。

Lock&Condition 实现的管程相对于 synchronized 实现的管程来说更加灵活、功能也更丰富。

结合我自己的经验，我认为了解原理比了解实现更能让你快速学好并发编程，所以没有介绍太多 Java SDK 并发包里锁和条件变量是如何实现的。但如果你对实现感兴趣，可以参考《Java 并发编程的艺术》一书的第 5 章《Java 中的锁》，里面详细介绍了实现原理，我觉得写得非常好。

另外，专栏里对 DefaultFuture 的代码缩减了很多，如果你感兴趣，也可以去看看完整版。Dubbo 的源代码在Github 上，DefaultFuture 的路径是：incubator-dubbo/dubbo-remoting/dubbo-remoting-api/src/main/java/org/apache/dubbo/remoting/exchange/support/DefaultFuture.java。

课后思考

DefaultFuture 里面唤醒等待的线程，用的是 signal()，而不是 signalAll()，你来分析一下，这样做是否合理呢?

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。

猜你喜欢

精选留言(4)

天涯煮酒

合理。

每个rpc请求都会占用一个线程并产生一个新的DefaultFuture实例，它们的lock&condition是不同的，并没有竞争关系

这里的lock&condition是用来做异步转同步的，使get()方法不必等待timeout那么久，用得很巧妙

6 2019-04-02

Binggle

这是一对一的关系，肯定只需要 signal。每个线程都是相互独立的，lock 和 condition 也是各自独享的。

2019-04-02

密码123456

不一定。如果这个类是单例，那就不合理。如果是一个实例对应一个请求，那就合理。

2019-04-02

张建磊

合理，等待条件都是response不空，等到通知后的动作都是返回response,也是通知一个线程。
老师，您在文中提到，子线程和新线程，代码上怎么区分呢？我认为在main中new thread,即使立刻返回main,也得在new thread之后。这是子线程还是新线程呢？

2019-04-02