## 题目:15、Lock和condition（下）：dubbo如何实现异步转同步？.md[](#15lockconditiondubbomd)

* **如何利用两个条件变量快速实现阻塞队列呢？**
* **同步与异步**
* **Dubbo源码分析**
* **总结**

Java SDK包里的lock有别于synchronized隐私锁的三个特性：能够响应中断、支持超时和非阻塞地获取锁。

这里主要讲解并发包里面的Conditiion，Condition的作用： **Condition实现了管程模型里面的条件变量**。

synchronize与Lock+Condition的另外一个区别：**synchronized只支持一个条件变量，而Lock&Condition实现的管程支持多个条件变量，这个也是二者的一个重要区别。**

## 如何利用两个条件变量快速实现阻塞队列呢？[](" \l "_1" \o "Permanent link)

一个阻塞队列，需要两个条件变量，一个是队列不空（孔队列不允许出队），另外一个是队列不满（队列已满不允许入队），如下：

**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**();**

**}**

*//省入队操作*

*//入队后，通知出队*

**}finally{**

lock**.**unlock**();**

**}**

**}**

*//出队*

**void** **deq(){**

lock**.**lock**();**

**try{**

**while(**队列已空**){**

*//等待队列不空*

notEmpty**.**await**();**

**}**

*//省出队操作*

*//出队后，通知入队*

**}finally{**

lock**.**unlock**();**

**}**

**}**

**}**

需要注意的是：Lock和Condition实现的管程，线程等待和通知需要调用await()、signal()、signalAll()，他们的语义和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。

## 同步和异步[](" \l "_2" \o "Permanent link)

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，如果需要等待结果，就是同步；如果不需要等待结果，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。

*// 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，可以通过下面的两种方式来实现：

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，创建一个新的现成饭执行主要逻辑，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[](" \l "dubbo" \o "Permanent link)

其实在编程领域，异步场景还是挺多的，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。

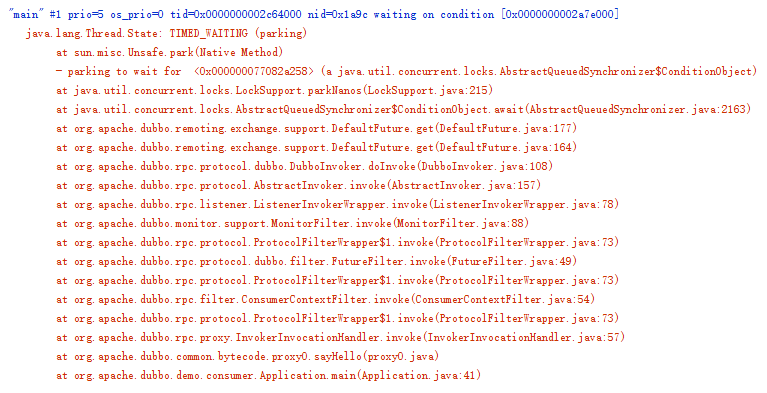
DemoService service **=** 初始化部分省略

String message **=** service**.**sayHello**(**"dubbo"**);**

System**.**out**.**println**(**message**);**

对于上面的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”); 线程会停下来等待结果。

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，但是调用线程却阻塞了，说明 Dubbo 帮我们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。



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。

**public** **class** **DubboInvoker{**

Result **doInvoke(**Invocation inv**){**

*//下面这行是源码中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。

*//创建锁与条件变量*

**private** **final** Lock lock **=** **new** ReentrantLock**()**

**private** **final** Condition done **=** lock**.**newCondition**();**

*//调用方通过该方法等待结果*

Object **get(int** timeout**){**

**long** start **=** Sytem**.**nanoTime**();**

lock**.**lock**();**

**try{**

**while(!**isDone**()){**

done**.**await**(**timeout**);**

**long** cur **=** System**.**nanoTime**();**

**if(**isDone**()||**cur**-**start**>**timeout**){**

**break;**

**}**

**}**

**}finally{**

lock**.**unlock**();**

**}**

**if(!**isDone**()){**

**throw** **new** timeoutException**();**

**}**

**return** **return** FromResponse**();**

**}**

*//RPC 结果是否已经返回*

**boolean** **isDone(){**

**return** response **!=null;**

**}**

*//RPC结果返回时调用该方法*

**private** **void** **doReceived(){**

lock**.**lock**();**

**try{**

response **=** res**;**

**if(**done**!=null){**

done**.**signal**()**

**}**

**}finally{**

lock**.**unlock**();**

**}**

**}**

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[](" \l "_3" \o "Permanent link)

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。

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。