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

2019-04-02 王宝令

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。

```
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。

```
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，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。

```
"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。

```
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。

```
// 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 = System.nanoTime();
  lock.lock();
  try {
 while (!isDone()) {
   done.await(timeout);
      long cur=System.nanoTime();
   if (isDone() ||
          cur-start > timeout){
     break;
   }
 }
  } finally {
  lock.unlock();
```

```
    lock.unlock();
    }
    if (!isDone()) {
   throw new TimeoutException();
    }
    return returnFromResponse();
  }
  // 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(Response res) {
    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

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。

精选留言

ZOU志伟 20

不合理，会导致很多请求超时，看了源码是调用signalAll()

2019-04-03

作者回复

写这一章的时候还是signal，后来有人提了个bug，就改成signalall了

2019-04-03

天涯煮酒 20

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

2019-04-02

张天屹 10

我理解异步的本质是利用多线程提升性能，异步一定是基于一个新开的线程，从调用线程来看是异步的，但是从新开的那个线程来看，正是同步（等待）的，只是对于调用方而言这种同步是透明的。正所谓生活哪有什么岁月静好，只是有人替你负重前行。

2019-04-04

作者回复

总结的太有文采了！异步加上非阻塞IO才有威力

2019-04-04

10buns 8

signal唤醒任意一个线程竞争锁，signalAll唤醒同一个条件变量的所有线程竞争锁。但都只有一个线程获得锁执行。区别只是被唤醒线程的数量。
所以用signalall可以避免极端情况线程只能等待超时，看了代码也是替代了signal

2019-04-04

约书亚 3

我有点不理解为什么这么多说合理的同学，Future这种类不应该经常由于用在闭包中，导致在多线程多上下文中传递嘛？如果我有多个线程都对同一个DefaultFuture实例调用get，而每个被唤醒的线程又不signal其他线程，那不就是只有一个线程最终会被唤醒，其他调用get的线程都是因为超时获取到的结果嘛？

2019-04-06

刘章周 2

回复：密码12345同学，如果是单例对象，response岂不是乱套了，每一个请求都对应自己的 r

esponse。另外singal()是合理的。因为每一个主线程对应一个子线程，不可能存在一个子线程对应多个请求。

2019-04-02

zhangtnty 2

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

作者回复

创建新线程和创建子线程没区别，都是约定俗成的说法而已

2019-04-03

Geek_e6f3ec 1

老师关于dubbo源码的执行流程有一点疑问。
以下是源码

```
// 调用通过该方法等待结果
Object get(int timeout){
long start = System.nanoTime();
lock.lock();
try{
while (!isDone()){
done.wait(timeout); // 在这里调用了等待方法后面的代码还能执行吗？ 我理解的管程，是在条件变量等待队列中阻塞等待，被唤醒之后也不是马上执行也要去管程入口等待队列，也就是lock.lock处等待获取锁。 老师是这样的吗？
long cur = System.nanoTime();
if (isDone()||cur-start> timeout){
break;
}
}
}finally {
lock.unlock();
}
return returnFromResponse();

}
```

2019-05-15

作者回复

会去获取锁，但是获取锁后，会执行wait后的代码

2019-05-15

牧名 1

DefaultFuture本质上只是一种future实现，所以理论上可以有多个线程同时持有同一个future并调用 get方法，如这时候使用signal()就有可能导致有些线程会请求超时

```java
DefaultFuture future = currentClient.request(inv, timeout);
for(int i=0; i< 10000; i++) {
new Thread(new Runnable() {
@Override
public void run() {
System.out.println(future.get().toString());
}
});
}
```

极客时间版权所有: https://time.geekbang.org/column/article/88487

2019-05-04

右耳听海 1

我看每个请求都会新建一个DefaultFuture，这个按道理应该只有一个线程阻塞，为什么需要signall

2019-04-28

右耳听海 1

in the method of org.apache.dubbo.remoting.exchange.support.DefaultFuture#doReceived, I think we should call done.signalAll() instead of done.signal() ,and it's unnecessary to check done != null because it's always true

2019-04-28

作者回复

留言这两点有同学都提到了。我表示震撼！

2019-04-28

W要改个网名 1

老师，有个疑问

为什么要判断done!=null呢？这个条件不是永远为true吗。

2019-04-03

作者回复

最新的代码他们已经改过来了

2019-04-03

ban 1

老师，求指教

DefaultFuturewhile这个类为什么要加 while(!isDone()) 这个条件，我看代码while里面加了done.await(timeout);是支持超时的，就是说设置5秒超时， if (isDone() || cur-start > timeout){，只要超过没有被signal()唤醒，那5秒就会自动唤醒，这时候就会在if (isDone() || cur-start > timeout){被校验通过，从而break，退出。这时候在加个while条件是不是没必要。
还是说加个while条件是因为时间到点的时候自动唤醒后，Response可能是空，而且时间cur-start > timeout 不超时，所以才有必要进行while再一次判断isDone()是否有值。

2019-04-03

作者回复

while条件是编程范式，可以回去看管程原理，搞工程要多重防护。超时后当然很有可能resp是空的

2019-04-03

QQ怪 1

我觉得很合理，因为每个请求都会实例化个DefactFeture,所以每个请求一个lock，明确知道需要唤醒哪个线程应该用asign()，同样这样做法也是为了应对高并发情况下的异步转同步需求吧！不知道对不对?

2019-04-02

liurh 1

老师，jdk已经实现了Future去把异步转换为同步，我们直接使用get()方法就会让线程阻塞的获取线程执行结果，为什么dubbo还要自己实现MESA模型，不太理解。业务中不知道什么时候该用这个模型

2019-04-02

作者回复

这个懂netty就知道了，一般工具类搞不定了就用它

2019-04-02

木刻 1

老师今天提到异步转同步，让我想到这两天看的zookeeper客户端源码，感觉应该也是这个机制，客户端同步模式下发送请求后会执行packet.wait，收到服务端响应后执行packet.notifyAll

2019-04-02

作者回复

2019-04-02

Binggle 1

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

作者回复

一对一的关系用signalall也不是不可以

2019-04-06

密码123456 1

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

Geek_89bbab

0

引用：

`

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。

`

请问老师，这里线程等待是否影响了并发呢？

2019-06-13

作者回复

影响，所以才有全异步的方案

2019-06-14

不瘦二十斤不换头像

谢特

0

demo那个类，加timeout有用吗，和不加有区别，怎么不都是一直死循环吗

2019-05-07