

15 | Nio2Endpoint组件：Tomcat如何实现异步I/O？

2019-06-13 李号双

深入拆解Tomcat & Jetty

[进入课程 >](#)



讲述：李号双

时长 11:44 大小 10.76M



我在专栏上一期里提到了 5 种 I/O 模型，相应的，Java 提供了 BIO、NIO 和 NIO.2 这些 API 来实现这些 I/O 模型。BIO 是我们最熟悉的同步阻塞，NIO 是同步非阻塞，那 NIO.2 又是什么呢？NIO 已经足够好了，为什么还要 NIO.2 呢？

NIO 和 NIO.2 最大的区别是，一个是同步一个是异步。我在上期提到过，异步最大的特点是，应用程序不需要自己去**触发**数据从内核空间到用户空间的**拷贝**。

为什么是应用程序去“触发”数据的拷贝，而不是直接从内核拷贝数据呢？这是因为应用程序是不能访问内核空间的，因此数据拷贝肯定是由内核来做，关键是谁来触发这个动作。

是内核主动将数据拷贝到用户空间并通知应用程序。还是等待应用程序通过 Selector 来查询，当数据就绪后，应用程序再发起一个 read 调用，这时内核再把数据从内核空间拷贝到

用户空间。

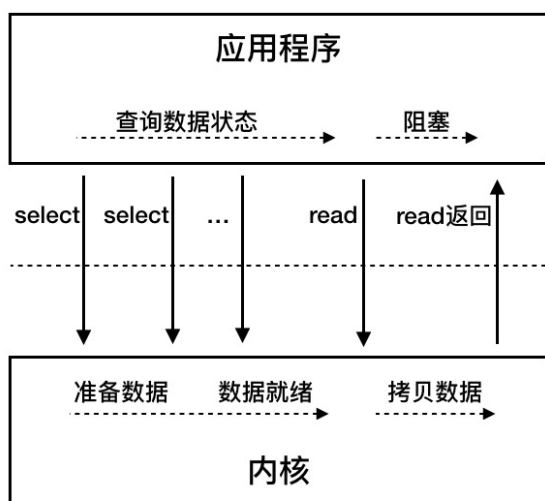
需要注意的是，数据从内核空间拷贝到用户空间这段时间，应用程序还是阻塞的。所以你会看到异步的效率是高于同步的，因为异步模式下应用程序始终不会被阻塞。下面我以网络数据读取为例，来说明异步模式的工作过程。

首先，应用程序在调用 read API 的同时告诉内核两件事情：数据准备好了以后拷贝到哪个 Buffer，以及调用哪个回调函数去处理这些数据。

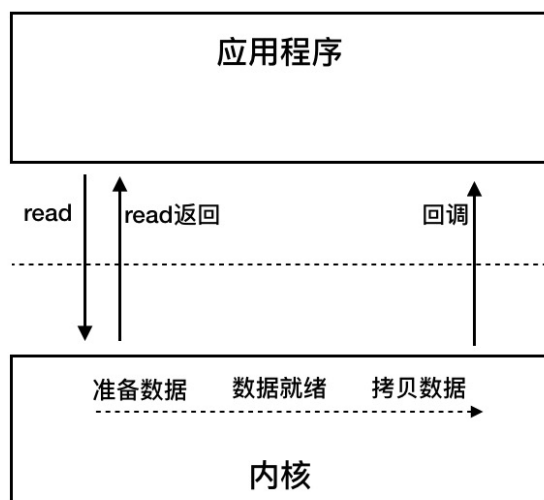
之后，内核接到这个 read 指令后，等待网卡数据到达，数据到了后，产生硬件中断，内核在中断程序里把数据从网卡拷贝到内核空间，接着做 TCP/IP 协议层面的数据解包和重组，再把数据拷贝到应用程序指定的 Buffer，最后调用应用程序指定的回调函数。

你可能通过下面这张图来回顾一下同步与异步的区别：

I/O多路复用（同步）



异步



我们可以看到在异步模式下，应用程序当了“甩手掌柜”，内核则忙前忙后，但最大限度提高了 I/O 通信的效率。Windows 的 IOCP 和 Linux 内核 2.6 的 AIO 都提供了异步 I/O 的支持，Java 的 NIO.2 API 就是对操作系统异步 I/O API 的封装。

Java NIO.2 回顾

今天我们会重点关注 Tomcat 是如何实现异步 I/O 模型的，但在这之前，我们先来简单回顾下如何用 Java 的 NIO.2 API 来编写一个服务端程序。

```
1 public class Nio2Server {
2
3     void listen(){
4         //1. 创建一个线程池
5         ExecutorService es = Executors.newCachedThreadPool();
6
7         //2. 创建异步通道群组
8         AsynchronousChannelGroup tg = AsynchronousChannelGroup.withCachedThreadPool(es, 1
9
10        //3. 创建服务端异步通道
11        AsynchronousServerSocketChannel assc = AsynchronousServerSocketChannel.open(tg);
12
13        //4. 绑定监听端口
14        assc.bind(new InetSocketAddress(8080));
15
16        //5. 监听连接，传入回调类处理连接请求
17        assc.accept(this, new AcceptHandler());
18    }
19 }
```

上面的代码主要做了 5 件事情：

1. 创建一个线程池，这个线程池用来执行来自内核的回调请求。
2. 创建一个 `AsynchronousChannelGroup`，并绑定一个线程池。
3. 创建 `AsynchronousServerSocketChannel`，并绑定到 `AsynchronousChannelGroup`。
4. 绑定一个监听端口。
5. 调用 `accept` 方法开始监听连接请求，同时传入一个回调类去处理连接请求。请你注意，`accept` 方法的第一个参数是 `this` 对象，就是 `Nio2Server` 对象本身，我在下文还会讲为什么要传入这个参数。

你可能会问，为什么需要创建一个线程池呢？其实在异步 I/O 模型里，应用程序不知道数据在什么时候到达，因此向内核注册回调函数，当数据到达时，内核就会调用这个回调函数。同时为了提高处理速度，会提供一个线程池给内核使用，这样不会耽误内核线程的工作，内核只需要把工作交给线程池就立即返回了。


我们再来看看处理连接的回调类 `AcceptHandler` 是什么样的。

```

1 //AcceptHandler 类实现了 CompletionHandler 接口的 completed 方法。它还有两个模板参数，第一个
2 public class AcceptHandler implements CompletionHandler<AsynchronousSocketChannel, Nio2S
3
4 // 具体处理连接请求的就是 completed 方法，它有两个参数：第一个是异步通道，第二个就是上面传入
5 @Override
6 public void completed(AsynchronousSocketChannel asc, Nio2Server attachment) {
7     // 调用 accept 方法继续接收其他客户端的请求
8     attachment.assc.accept(attachment, this);
9
10    //1. 先分配好 Buffer，告诉内核，数据拷贝到哪里去
11    ByteBuffer buf = ByteBuffer.allocate(1024);
12
13    //2. 调用 read 函数读取数据，除了把 buf 作为参数传入，还传入读回调类
14    channel.read(buf, buf, new ReadHandler(asc));
15
16 }

```

我们看到它实现了 CompletionHandler 接口，下面我们先来看看 CompletionHandler 接口的定义。

 复制代码

```

1 public interface CompletionHandler<V,A> {
2
3     void completed(V result, A attachment);
4
5     void failed(Throwable exc, A attachment);
6 }

```

CompletionHandler 接口有两个模板参数 V 和 A，分别表示 I/O 调用的返回值和附件类。比如 accept 的返回值就是 AsynchronousSocketChannel，而附件类由用户自己决定，在 accept 的调用中，我们传入了一个 Nio2Server。因此 AcceptHandler 带有了两个模板参数：AsynchronousSocketChannel 和 Nio2Server。

CompletionHandler 有两个方法：completed 和 failed，分别在 I/O 操作成功和失败时调用。completed 方法有两个参数，其实就是前面说的两个模板参数。也就是说，Java 的 NIO.2 在调用回调方法时，会把返回值和附件类当作参数传给 NIO.2 的使用者。

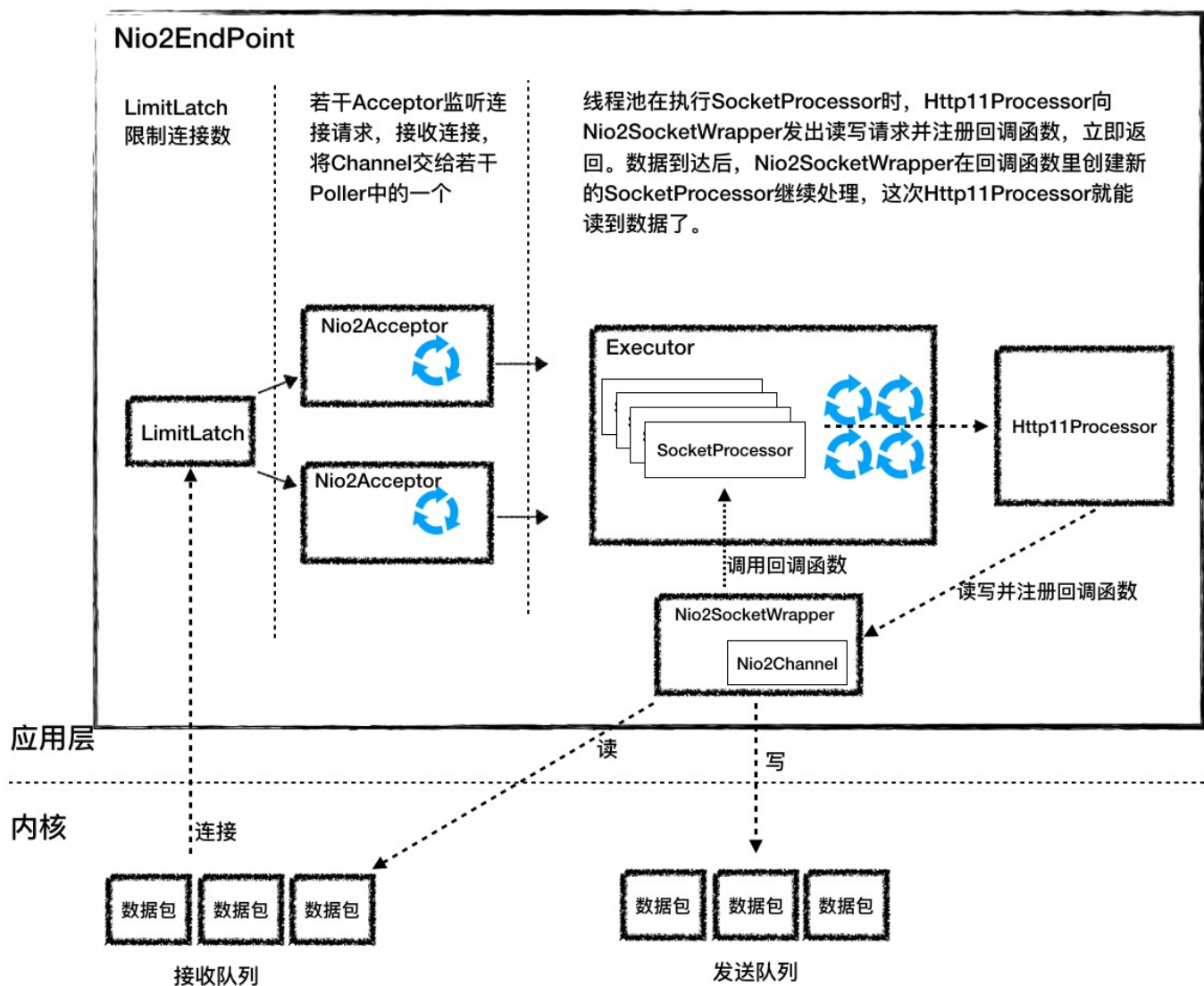
下面我们再来看看处理读的回调类 ReadHandler 长什么样子。

```
1 public class ReadHandler implements CompletionHandler<Integer, ByteBuffer> {  
2     // 读取到消息后的处理  
3     @Override  
4     public void completed(Integer result, ByteBuffer attachment) {  
5         //attachment 就是数据，调用 flip 操作，其实就是把读的位置移动最前面  
6         attachment.flip();  
7         // 读取数据  
8         ...  
9     }  
10  
11     void failed(Throwable exc, A attachment){  
12         ...  
13     }  
14 }
```

read 调用的返回值是一个整型数，所以我们回调方法里的第一个参数就是一个整型，表示有多少数据被读取到了 Buffer 中。第二个参数是一个 ByteBuffer，这是因为我们在调用 read 方法时，把用来存放数据的 ByteBuffer 当作附件类传进去了，所以在回调方法里，有 ByteBuffer 类型的参数，我们直接从这个 ByteBuffer 里获取数据。

Nio2Endpoint

掌握了 Java NIO.2 API 的使用以及服务端程序的工作原理之后，再来理解 Tomcat 的异步 I/O 实现就不难了。我们先通过一张图来看看 Nio2Endpoint 有哪些组件。



从图上看，总体工作流程跟 NioEndpoint 是相似的。

LimitLatch 是连接控制器，它负责控制最大连接数。

Nio2Acceptor 扩展了 Acceptor，用异步 I/O 的方式来接收连接，跑在一个单独的线程里，也是一个线程组。Nio2Acceptor 接收新的连接后，得到一个 `AsynchronousSocketChannel`，Nio2Acceptor 把 `AsynchronousSocketChannel` 封装成一个 `Nio2SocketWrapper`，并创建一个 `SocketProcessor` 任务类交给线程池处理，并且 `SocketProcessor` 持有 `Nio2SocketWrapper` 对象。

Executor 在执行 `SocketProcessor` 时，`SocketProcessor` 的 `run` 方法会调用 `Http11Processor` 来处理请求，`Http11Processor` 会通过 `Nio2SocketWrapper` 读取和解析请求数据，请求经过容器处理后，再把响应通过 `Nio2SocketWrapper` 写出。

需要你注意 Nio2Endpoint 跟 NioEndpoint 的一个明显不同点是，**Nio2Endpoint 中没有 Poller 组件，也就是没有 Selector。这是为什么呢？因为在异步 I/O 模式下，Selector 的工作交给内核来做了。**

接下来我详细介绍一下 Nio2Endpoint 各组件的设计。

Nio2Acceptor

和 NioEndpoint 一样，Nio2Endpoint 的基本思路是用 LimitLatch 组件来控制连接数，但是 Nio2Acceptor 的监听连接的过程不是在一个死循环里不断的调 accept 方法，而是通过回调函数来完成的。我们来看看它的连接监听方法：

 复制代码

```
1 serverSock.accept(null, this);
```

其实就是调用了 accept 方法，注意它的第二个参数是 this，表明 Nio2Acceptor 自己就是处理连接的回调类，因此 Nio2Acceptor 实现了 CompletionHandler 接口。那么它是如何实现 CompletionHandler 接口的呢？

 复制代码

```
1 protected class Nio2Acceptor extends Acceptor<AsynchronousSocketChannel>
2     implements CompletionHandler<AsynchronousSocketChannel, Void> {
3
4     @Override
5     public void completed(AsynchronousSocketChannel socket,
6         Void attachment) {
7
8         if (isRunning() && !isPaused()) {
9             if (getMaxConnections() == -1) {
10                 // 如果没有连接限制，继续接收新的连接
11                 serverSock.accept(null, this);
12             } else {
13                 // 如果有连接限制，就在线程池里跑 Run 方法，Run 方法会检查连接数
14                 getExecutor().execute(this);
15             }
16             // 处理请求
17             if (!setSocketOptions(socket)) {
18                 closeSocket(socket);
19             }
20         }
```

可以看到 CompletionHandler 的两个模板参数分别是 AsynchronousServerSocketChannel 和 Void，我在前面说过第一个参数就是 accept 方法的返回值，第二个参数是附件类，由用户自己决定，这里为 Void。completed 方法的处理逻辑比较简单：

如果没有连接限制，继续在本线程中调用 accept 方法接收新的连接。

如果有连接限制，就在线程池里跑 run 方法去接收新的连接。那为什么要跑 run 方法呢，因为在 run 方法里会检查连接数，当连接达到最大数时，线程可能会被 LimitLatch 阻塞。为什么要放在线程池里跑呢？这是因为如果放在当前线程里执行，completed 方法可能被阻塞，会导致这个回调方法一直不返回。

接着 completed 方法会调用 setSocketOptions 方法，在这个方法里，会创建 Nio2SocketWrapper 和 SocketProcessor，并交给线程池处理。

Nio2SocketWrapper


Nio2SocketWrapper 的主要作用是封装 Channel，并提供接口给 Http11Processor 读写数据。讲到这里你是不是有个疑问：Http11Processor 是不能阻塞等待数据的，按照异步 I/O 的套路，Http11Processor 在调用 Nio2SocketWrapper 的 read 方法时需要注册回调类，read 调用会立即返回，问题是立即返回后 Http11Processor 还没有读到数据，怎么办呢？这个请求的处理不就失败了吗？

为了解决这个问题，Http11Processor 是通过 2 次 read 调用来完成数据读取操作的。

第一次 read 调用：连接刚刚建立好后，Acceptor 创建 SocketProcessor 任务类交给线程池去处理，Http11Processor 在处理请求的过程中，会调用 Nio2SocketWrapper 的 read 方法发出第一次读请求，同时注册了回调类 readCompletionHandler，因为数据没读到，Http11Processor 把当前的 Nio2SocketWrapper 标记为数据不完整。**接着 SocketProcessor 线程被回收，Http11Processor 并没有阻塞等待数据。**这里请注意，Http11Processor 维护了一个 Nio2SocketWrapper 列表，也就是维护了连接的状态。

第二次 read 调用：当数据到达后，内核已经把数据拷贝到 Http11Processor 指定的 Buffer 里，同时回调类 readCompletionHandler 被调用，在这个回调处理方法里会**重新创建一个新的 SocketProcessor 任务来继续处理这个连接**，而这个新的 SocketProcessor 任务类持有原来那个 Nio2SocketWrapper，这一次 Http11Processor 可以通过 Nio2SocketWrapper 读取数据了，因为数据已经到了应用层的 Buffer。

这个回调类 readCompletionHandler 的源码如下，最关键的一点是，**Nio2SocketWrapper 是作为附件类来传递的**，这样在回调函数里能拿到所有的上下文。

 复制代码

```
1 this.readCompletionHandler = new CompletionHandler<Integer, SocketWrapperBase<Nio2Channel>>() {
2     public void completed(Integer nBytes, SocketWrapperBase<Nio2Channel> attachment) {
3         ...
4         // 通过附件类 SocketWrapper 拿到所有的上下文
5         Nio2SocketWrapper.this.getEndpoint().processSocket(attachment, SocketEvent.OPEN_
6     }
7
8     public void failed(Throwable exc, SocketWrapperBase<Nio2Channel> attachment) {
9         ...
10    }
11 }
```

本期精华

在异步 I/O 模型里，内核做了很多事情，它把数据准备好，并拷贝到用户空间，再通知应用程序去处理，也就是调用应用程序注册的回调函数。Java 在操作系统 异步 IO API 的基础上进行了封装，提供了 Java NIO.2 API，而 Tomcat 的异步 I/O 模型就是基于 Java NIO.2 实现的。

由于 NIO 和 NIO.2 的 API 接口和使用方法完全不同，可以想象一个系统中如果已经支持同步 I/O，要再支持异步 I/O，改动是比较大的，很有可能不得不重新设计组件之间的接口。但是 Tomcat 通过充分的抽象，比如 SocketWrapper 对 Channel 的封装，再加上 Http11Processor 的两次 read 调用，巧妙地解决了这个问题，使得协议处理器 Http11Processor 和 I/O 通信处理器 Endpoint 之间的接口保持不变。

课后思考

我在文章开头介绍 Java NIO.2 的使用时，提到过要创建一个线程池来处理异步 I/O 的回调，那么这个线程池跟 Tomcat 的工作线程池 Executor 是同一个吗？如果不是，它们有什么关系？

不知道今天的内容你消化得如何？如果还有疑问，请大胆的在留言区提问，也欢迎你把你的课后思考和心得记录下来，与我和其他同学一起讨论。如果你觉得今天有所收获，欢迎你把它分享给你的朋友。



深入拆解 Tomcat & Jetty

从源码角度深度探索 Java 中间件

李号双

eBay 技术主管



新版升级：点击「👤 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 14 | NioEndpoint组件：Tomcat如何实现非阻塞I/O？

下一篇 16 | AprEndpoint组件：Tomcat APR提高I/O性能的秘密

精选留言 (17)

写留言



西兹兹

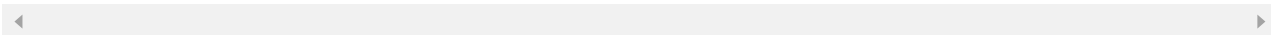
2019-06-15

1

李老师好，请问nio1，tomcat里nio为什么不参考netty，通过使用堆外内存来避免零拷贝问题？

展开 ▾

作者回复: 主要还是堆外内存管理起来没有JVM堆那么方便, 为了稳定性的考虑吧, 另外APR就是堆外内存的方案, 也就是已经提供了这个选项。



西兹兹

2019-06-15

👍 1

李老师好, 这节精彩, 特别是两次read讲的用心。

1 nio2图里确实有个poller字样

2 思考题回答, 个人认为第二次read用的是work线程池, 因为内核已经准备好完整开箱可用的数据, 直接使用即可, 无须过多的线程上下文切换。

展开 ▾



yungoo

2019-06-13

👍 1

我所看的tomcat 8.5的代码跟专栏所讲已经有些不一致了。已经没有Nio2Acceptor了, accept获取连接用的是Future accept()。

展开 ▾

作者回复: 我使用的是最新版的代码:

<https://github.com/apache/tomcat/blob/master/java/org/apache/tomcat/util/net/Nio2Endpoint.java>



z.l

2019-06-15

👍

linux没有真正实现异步IO, 所以linux环境下NIO和NIO2的性能差别是不是不大?



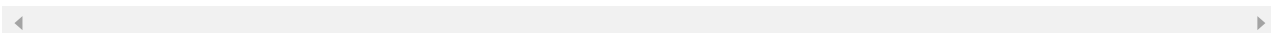
helloWorld

2019-06-15

👍

老师以后在文章中的例子可以给出完整的代码吗?

作者回复: 我尽量贴全, 有时候代码太多不好贴, 最好课后去看看源码





802.11

2019-06-15



Http11Processor的2次read是在哪个类中呢，没有找到。。。

作者回复: Nio2SocketWrapper的read方法，这个方法会被调用两次，不是串行调两次，而是Poller会先后创建两个SocketProcessor任务类，在两个线程中执行，执行过程中每次Http11Processor都会调Nio2SocketWrapper的read方法。

```
public int read(boolean block, ByteBuffer to){
```

```
//第二次调用时直接通过这个方法取数据
```

```
int nRead = populateReadBuffer(to);
```

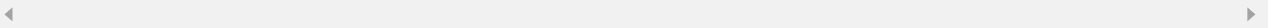
```
...
```

```
//第一次时数据没取到，会调用下面这个方法去真正执行I/O操作并注册回调函数：
```

```
nRead = fillReadBuffer(block);
```

```
...
```

```
}
```



飞翔

2019-06-14



老师 有了nio2endpoint是不是就没人用nioendpoint了？



飞翔

2019-06-14

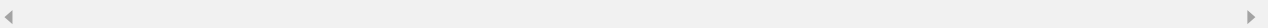


老师tomcat 在哪里配置 使用nioendpoint 还是nio2endpoint，能否给个例子

作者回复: server.xml中：

```
<Connector port="8443" protocol="org.apache.coyote.http11.Http11NioProtocol"
    maxThreads="150" SSLEnabled="true">
```

```
</Connector>
```





802.11

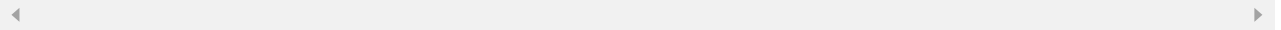
2019-06-14



老师，为什么IO操作的返回值第一个参数是channel呢，不理解。按照描述，既然是返回值应该是一些具体的值吧，channel为什么会出现在这里呢。

展开 ▾

作者回复: accept调用返回channel对象



binginx

2019-06-14



老师，tomcat使用异步nio2比使用nio性能上提高很大吗？



发条橙子 ...

2019-06-14



老师，这两张讲的I/O有点难啃，主要还是底子太薄。反复看了几遍有几个疑问点希望老师指点一下 😊

1. read请求是怎么发出来的 是通过调用select方法发出来的么？

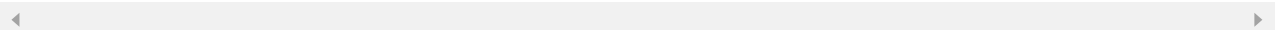
...

展开 ▾

作者回复: 1，select只是查询，真正发出read调用的还是read方法

2，好像没有异步阻塞这个说法

3，同步阻塞模型，read调用发起时，数据可能还没到网卡。如果io多路复用，read调用时，数据已经到了内核空间，因为之前select已经查到数据到了，应用才调read



nihil

2019-06-13



问下老师，这个Tomcat这个IO模型是将数据拷贝了两次么，还是有做特殊优化

作者回复: 数据没有拷贝两次, 第一次read调用是读不到数据的, 因为这个时候数据还没应用层的Buffer, 只是注册一个回调函数, 当内核将数据拷贝到了应用层Buffer, 调用回调函数, 在回调函数里, HttpProcessor再发起一次read, read方法首先会检查数据是不是已经到了Buffer, 如果是, 直接读取Buffer返回, 这一次并没有真正向内核发起read调用。



W.T

2019-06-13



李老师实力派! 👍



木偶人King

2019-06-13



前边看的比较顺畅。今天的没太看懂。顿生挫败感。

加油加油。只能多看几遍了



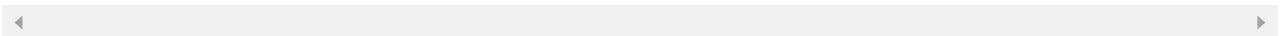
半斤八两

2019-06-13



老师图片上Acceptor的行为还是交给poller, 是不是有问题啊.....还是我理解错了.....

作者回复: 这一篇的图片上没有poller吧



2019-06-13



@WL Linux 的内存映射了解下



WL

2019-06-13



老师关于IO模型内存我有两个问题:

1. 配内存的时候, 是不是因为堆内存会受到GC的影响导致地址变化, 所以不能直接使用不能使用堆内存, 如果使用堆内存的话也需要先指向一个固定的堆外内存, 所以使用堆外内

存就可以避免GC对内存地址的影响。

2. 是不是IO在读数据的时候经过两次数据拷贝，从网卡到内核态，从内核态到用户态， ...

展开 ∨

作者回复: 好问题，下一篇就会解释这个问题。

