

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

2019-06-11 李号双

深入拆解Tomcat & Jetty

[进入课程 >](#)



讲述：李号双

时长 12:44 大小 11.68M



UNIX 系统下的 I/O 模型有 5 种：同步阻塞 I/O、同步非阻塞 I/O、I/O 多路复用、信号驱动 I/O 和异步 I/O。这些名词我们好像都似曾相识，但这些 I/O 通信模型有什么区别？同步和阻塞似乎是一回事，到底有什么不同？等一下，在这之前你是不是应该问自己一个终极问题：什么是 I/O？为什么需要这些 I/O 模型？

所谓的**I/O 就是计算机内存与外部设备之间拷贝数据的过程**。我们知道 CPU 访问内存的速度远远高于外部设备，因此 CPU 是先把外部设备的数据读到内存里，然后再进行处理。请考虑一下这个场景，当你的程序通过 CPU 向外部设备发出一个读指令时，数据从外部设备拷贝到内存往往需要一段时间，这个时候 CPU 没事干了，你的程序是主动把 CPU 让给别人？还是让 CPU 不停地查：数据到了吗，数据到了吗.....

这就是 I/O 模型要解决的问题。今天我会先说说各种 I/O 模型的区别，然后重点分析 Tomcat 的 NioEndpoint 组件是如何实现非阻塞 I/O 模型的。

## Java I/O 模型

对于一个网络 I/O 通信过程，比如网络数据读取，会涉及两个对象，一个是调用这个 I/O 操作的用户线程，另外一个就是操作系统内核。一个进程的地址空间分为用户空间和内核空间，用户线程不能直接访问内核空间。

当用户线程发起 I/O 操作后，网络数据读取操作会经历两个步骤：

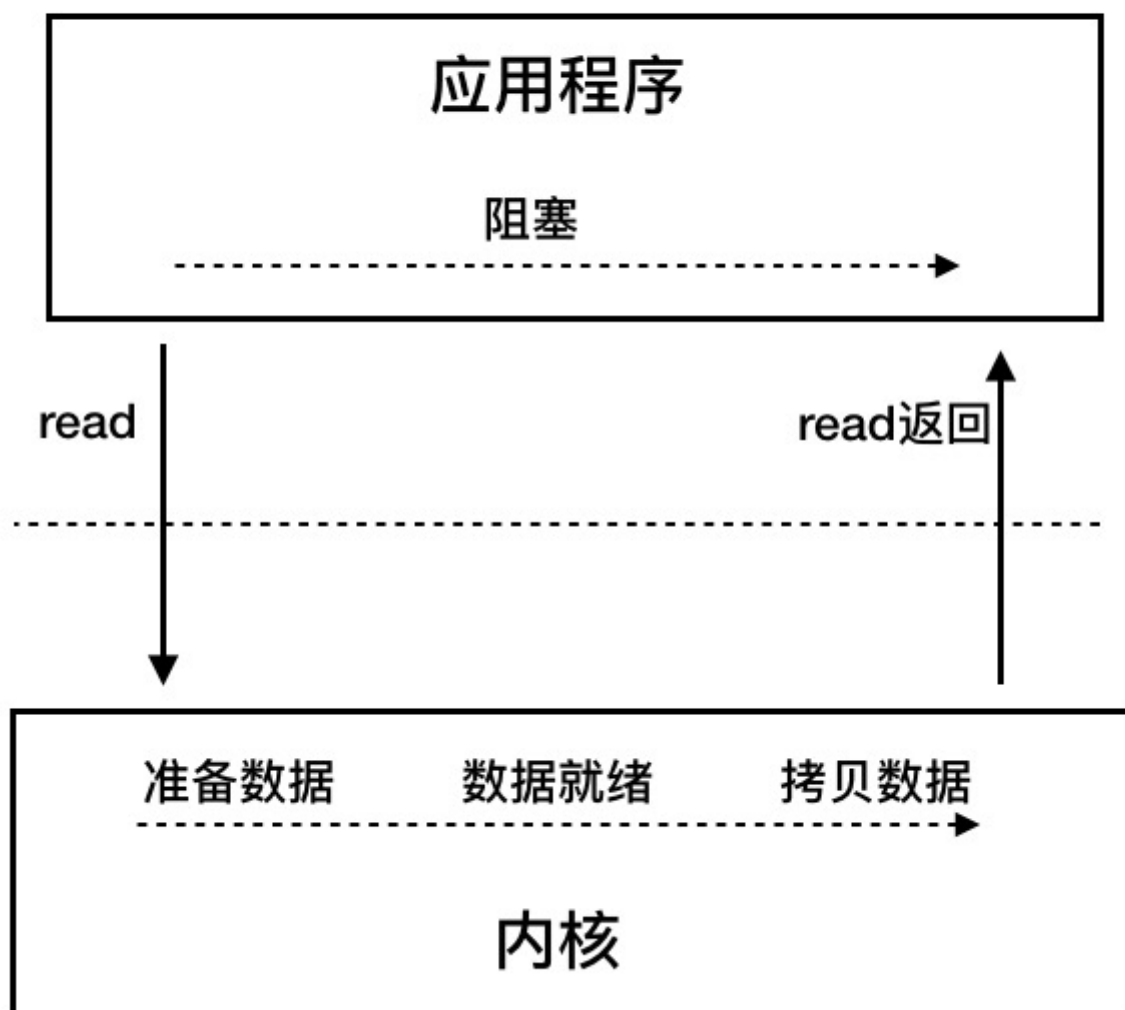
**用户线程等待内核将数据从网卡拷贝到内核空间。**

**内核将数据从内核空间拷贝到用户空间。**

各种 I/O 模型的区别就是：它们实现这两个步骤的方式是不一样的。

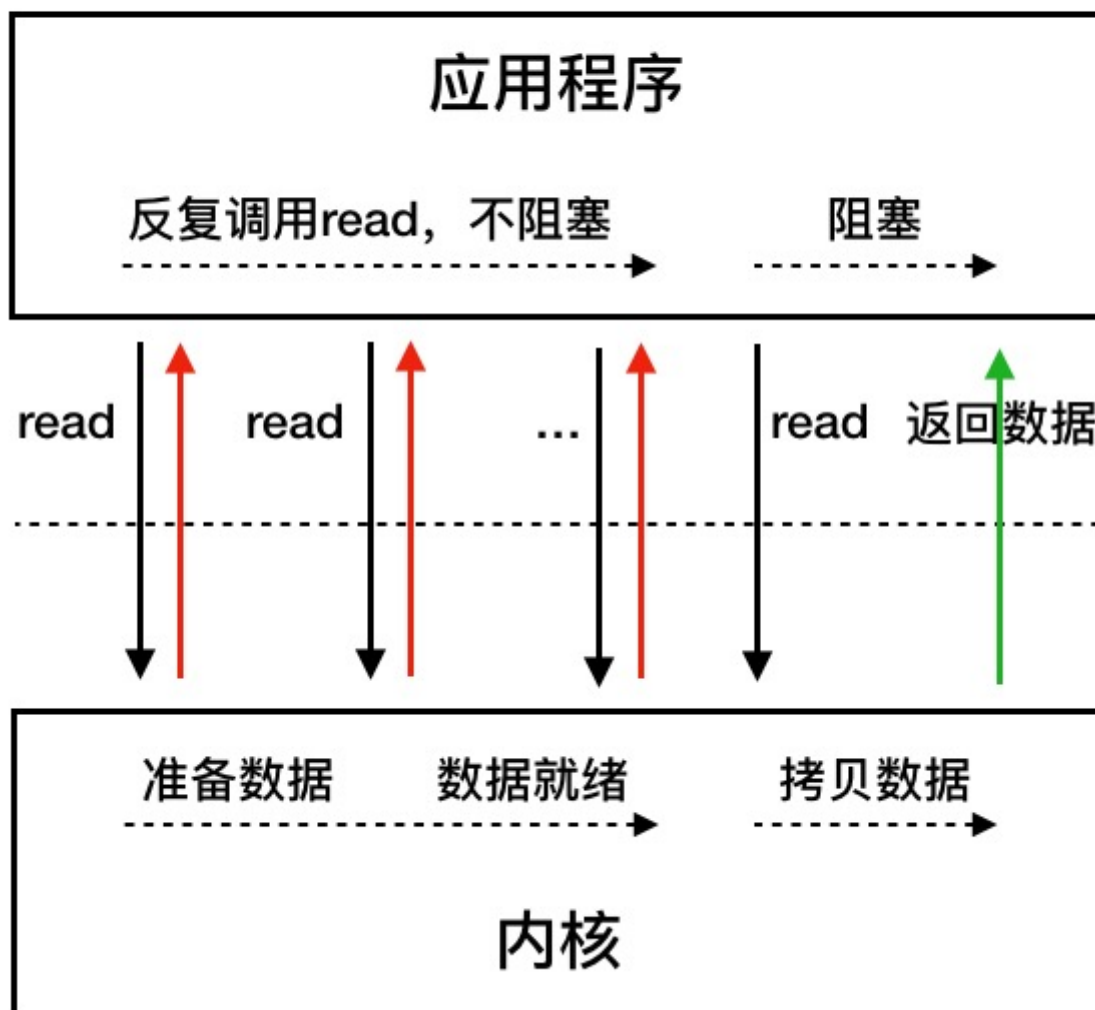
**同步阻塞 I/O**：用户线程发起 read 调用后就阻塞了，让出 CPU。内核等待网卡数据到来，把数据从网卡拷贝到内核空间，接着把数据拷贝到用户空间，再把用户线程叫醒。

# 同步阻塞



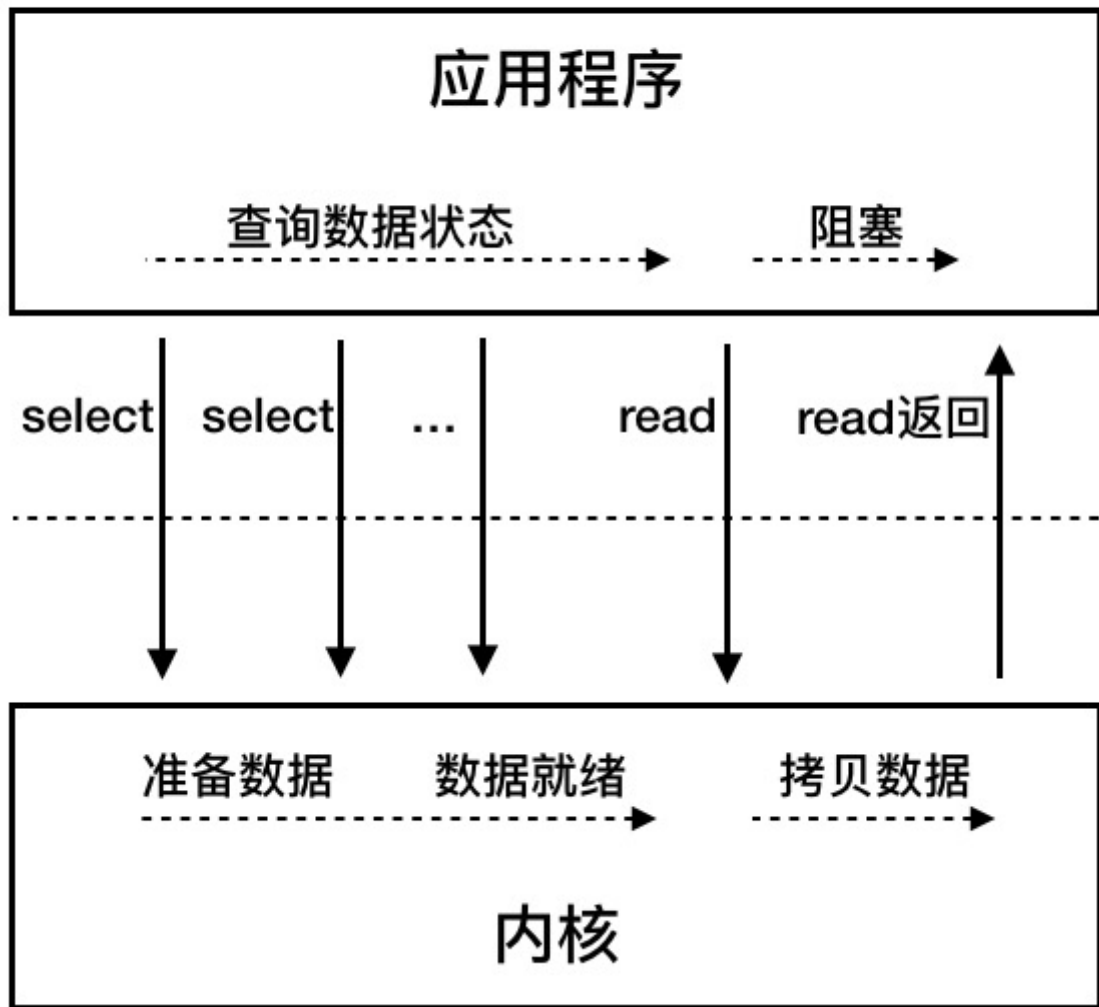
**同步非阻塞 I/O**：用户线程不断的发起 read 调用，数据没到内核空间时，每次都返回失败，直到数据到了内核空间，这一次 read 调用后，在等待数据从内核空间拷贝到用户空间这段时间里，线程还是阻塞的，等数据到了用户空间再把线程叫醒。

# 同步非阻塞



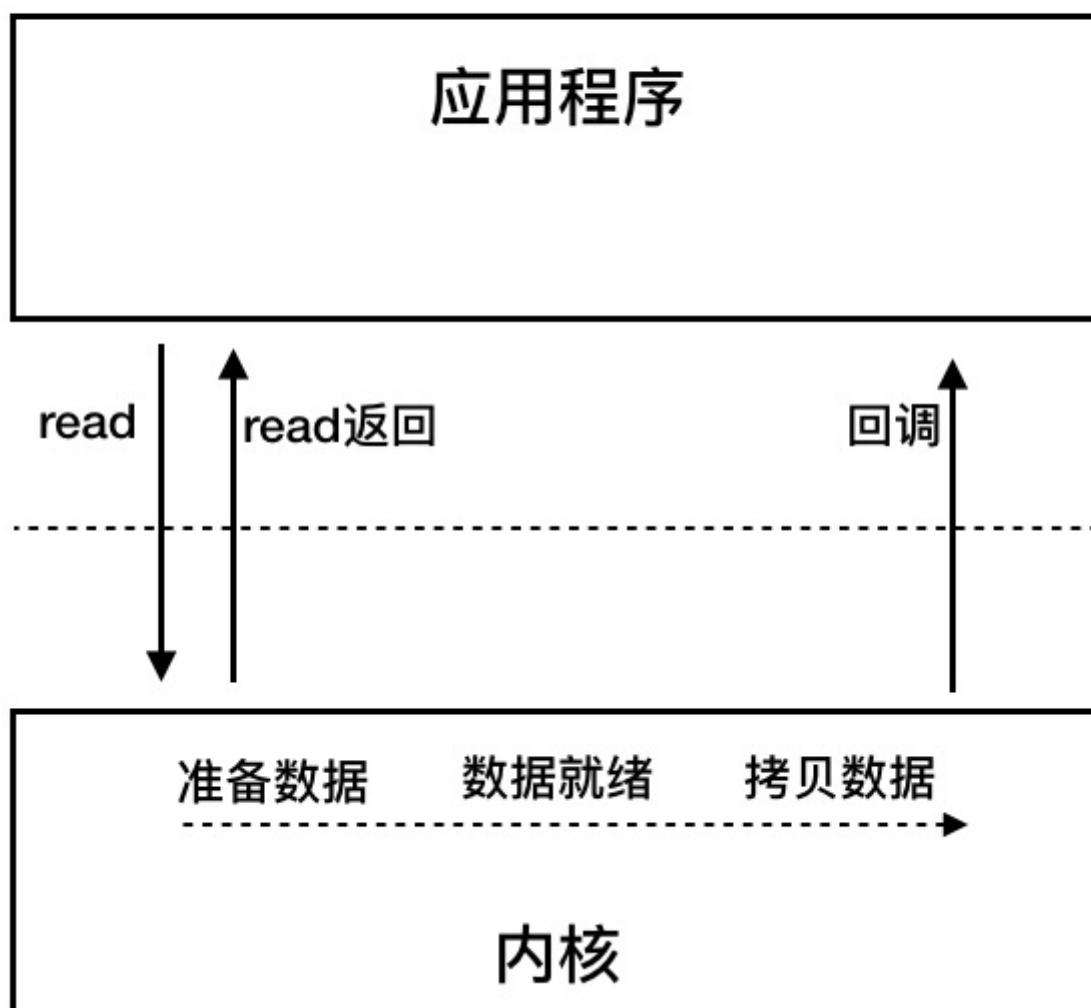
**I/O 多路复用：**用户线程的读取操作分成两步了，线程先发起 select 调用，目的是问内核数据准备好了吗？等内核把数据准备好了，用户线程再发起 read 调用。在等待数据从内核空间拷贝到用户空间这段时间里，线程还是阻塞的。那什么叫 I/O 多路复用呢？因为一次 select 调用可以向内核查多个数据通道（Channel）的状态，所以叫多路复用。

# I/O多路复用



**异步 I/O**：用户线程发起 read 调用的同时注册一个回调函数，read 立即返回，等内核将数据准备好后，再调用指定的回调函数完成处理。在这个过程中，用户线程一直没有阻塞。

# 异步



## NioEndpoint 组件

Tomcat 的 NioEndPoint 组件实现了 I/O 多路复用模型，接下来我会介绍 NioEndpoint 的实现原理，下一期我会介绍 Tomcat 如何实现异步 I/O 模型。

## 总体工作流程

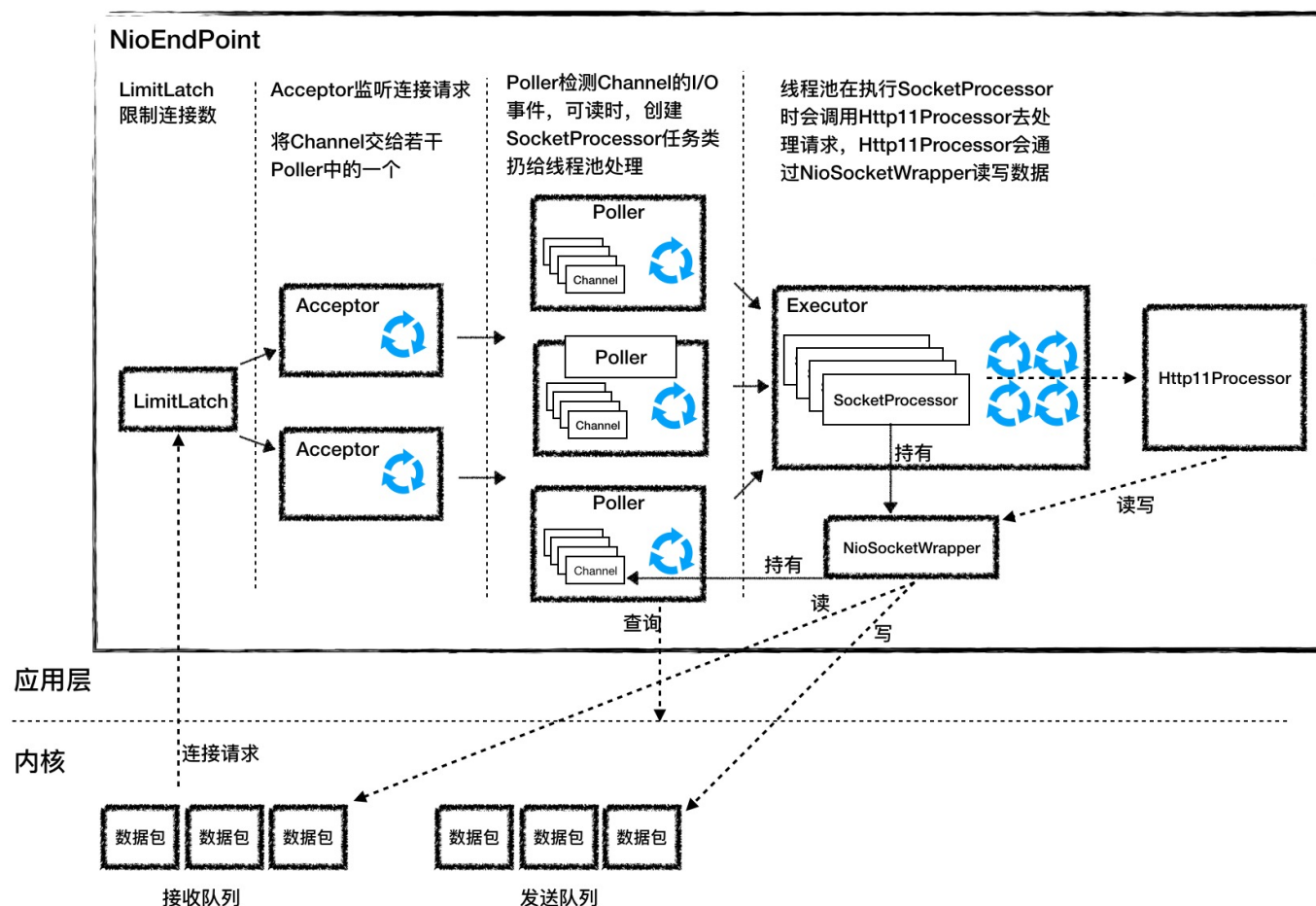
我们知道，对于 Java 的多路复用器的使用，无非是两步：

1. 创建一个 Selector，在它身上注册各种感兴趣的事件，然后调用 select 方法，等待感兴趣的事情发生。



2. 感兴趣的事情发生了，比如可以读了，这时便创建一个新的线程从 Channel 中读数据。

Tomcat 的 NioEndpoint 组件虽然实现比较复杂，但基本原理就是上面两步。我们先来看看它有哪些组件，它一共包含 LimitLatch、Acceptor、Poller、SocketProcessor 和 Executor 共 5 个组件，它们的工作过程如下图所示。



LimitLatch 是连接控制器，它负责控制最大连接数，NIO 模式下默认是 10000，达到这个阈值后，连接请求被拒绝。

Acceptor 跑在一个单独的线程里，它在一个死循环里调用 accept 方法来接收新连接，一旦有新的连接请求到来，accept 方法返回一个 Channel 对象，接着把 Channel 对象交给 Poller 去处理。


Poller 的本质是一个 Selector，也跑在单独线程里。Poller 在内部维护一个 Channel 数组，它在一个死循环里不断检测 Channel 的数据就绪状态，一旦有 Channel 可读，就生成一个 SocketProcessor 任务对象扔给 Executor 去处理。

Executor 就是线程池，负责运行 SocketProcessor 任务类，SocketProcessor 的 run 方法会调用 Http11Processor 来读取和解析请求数据。我们知道，Http11Processor 是应用层协议的封装，它会调用容器获得响应，再把响应通过 Channel 写出。

接下来我详细介绍一下各组件的设计特点。

## LimitLatch

LimitLatch 用来控制连接个数，当连接数到达最大时阻塞线程，直到后续组件处理完一个连接后将连接数减 1。请你注意到最大连接数后操作系统底层还是会接收客户端连接，但用户层已经不再接收。LimitLatch 的核心代码如下：

 复制代码

```
1 public class LimitLatch {
2     private class Sync extends AbstractQueuedSynchronizer {
3
4         @Override
5         protected int tryAcquireShared() {
6             long newCount = count.incrementAndGet();
7             if (newCount > limit) {
8                 count.decrementAndGet();
9                 return -1;
10            } else {
11                return 1;
12            }
13        }
14
15        @Override
16        protected boolean tryReleaseShared(int arg) {
17            count.decrementAndGet();
18            return true;
19        }
20    }
21
22    private final Sync sync;
23    private final AtomicLong count;
24    private volatile long limit;
25
26    // 线程调用这个方法来获得接收新连接的许可，线程可能被阻塞
27    public void countUpOrAwait() throws InterruptedException {
28        sync.acquireSharedInterruptibly(1);
29    }
30
31    // 调用这个方法来释放一个连接许可，那么前面阻塞的线程可能被唤醒
32    public long countDown() {
33        sync.releaseShared(0);
```



```
34         long result = getCount();
35         return result;
36     }
37 }
```

从上面的代码我们看到，LimitLatch 内部定义了内部类 Sync，而 Sync 扩展了 AQS，AQS 是 Java 并发包中的一个核心类，它在内部维护一个状态和一个线程队列，可以用来**控制线程什么时候挂起，什么时候唤醒**。我们可以扩展它来实现自己的同步器，实际上 Java 并发包里的锁和条件变量等等都是通过 AQS 来实现的，而这里的 LimitLatch 也不例外。


理解上面的代码时有两个要点：

1. 用户线程通过调用 LimitLatch 的 countUpOrAwait 方法来拿到锁，如果暂时无法获取，这个线程会被阻塞到 AQS 的队列中。那 AQS 怎么知道是阻塞还是不阻塞用户线程呢？其实这是由 AQS 的使用者来决定的，也就是内部类 Sync 来决定的，因为 Sync 类重写了 AQS 的**tryAcquireShared() 方法**。它的实现逻辑是如果当前连接数 count 小于 limit，线程能获取锁，返回 1，否则返回 -1。
2. 如何用户线程被阻塞到了 AQS 的队列，那什么时候唤醒呢？同样是由 Sync 内部类决定，Sync 重写了 AQS 的**releaseShared() 方法**，其实就是当一个连接请求处理完了，这时又可以接收一个新连接了，这样前面阻塞的线程将会被唤醒。

其实你会发现 AQS 就是一个骨架抽象类，它帮我们搭了个架子，用来控制线程的阻塞和唤醒。具体什么时候阻塞、什么时候唤醒由你来决定。我们还注意到，当前线程数被定义成原子变量 AtomicLong，而 limit 变量用 volatile 关键字来修饰，这些并发编程的实际运用。

## Acceptor

Acceptor 实现了 Runnable 接口，因此可以跑在单独线程里。一个端口号只能对应一个 ServerSocketChannel，因此这个 ServerSocketChannel 是在多个 Acceptor 线程之间共享的，它是 Endpoint 的属性，由 Endpoint 完成初始化和端口绑定。初始化过程如下：

 复制代码

```
1 serverSock = ServerSocketChannel.open();
```

```
2 serverSock.socket().bind(addr, getAcceptCount());
3 serverSock.configureBlocking(true);
```

从上面的初始化代码我们可以看到两个关键信息：

1. bind 方法的第二个参数表示操作系统的等待队列长度，我在上面提到，当应用层面的连接数到达最大值时，操作系统可以继续接收连接，那么操作系统能继续接收的最大连接数就是这个队列长度，可以通过 acceptCount 参数配置，默认是 100。

2. ServerSocketChannel 被设置成阻塞模式，也就是说它是以阻塞的方式接收连接的。

ServerSocketChannel 通过 accept() 接受新的连接，accept() 方法返回获得 SocketChannel 对象，然后将 SocketChannel 对象封装在一个 PollerEvent 对象中，并将 PollerEvent 对象压入 Poller 的 Queue 里，这是个典型的生产者 - 消费者模式，Acceptor 与 Poller 线程之间通过 Queue 通信。

## Poller

Poller 本质是一个 Selector，它内部维护一个 Queue，这个 Queue 定义如下：

 复制代码

```
1 private final SynchronizedQueue<PollerEvent> events = new SynchronizedQueue<>();
```

SynchronizedQueue 的方法比如 offer、poll、size 和 clear 方法，都使用了 Synchronized 关键字进行修饰，用来保证同一时刻只有一个 Acceptor 线程对 Queue 进行读写。同时有多个 Poller 线程在运行，每个 Poller 线程都有自己的 Queue。每个 Poller 线程可能同时被多个 Acceptor 线程调用来注册 PollerEvent。同样 Poller 的个数可以通过 pollers 参数配置。

Poller 不断的通过内部的 Selector 对象向内核查询 Channel 的状态，一旦可读就生成任务类 SocketProcessor 交给 Executor 去处理。Poller 的另一个重要任务是循环遍历检查自己所管理的 SocketChannel 是否已经超时，如果有超时就关闭这个 SocketChannel。

## SocketProcessor

我们知道，Poller 会创建 SocketProcessor 任务类交给线程池处理，而 SocketProcessor 实现了 Runnable 接口，用来定义 Executor 中线程所执行的任务，主要就是调用 Http11Processor 组件来处理请求。Http11Processor 读取 Channel 的数据来生成 ServletRequest 对象，这里请你注意：

Http11Processor 并不是直接读取 Channel 的。这是因为 Tomcat 支持同步非阻塞 I/O 模型和异步 I/O 模型，在 Java API 中，相应的 Channel 类也是不一样的，比如有 AsynchronousSocketChannel 和 SocketChannel，为了对 Http11Processor 屏蔽这些差异，Tomcat 设计了一个包装类叫作 SocketWrapper，Http11Processor 只调用 SocketWrapper 的方法去读写数据。

## Executor

Executor 是 Tomcat 定制版的线程池，它负责创建真正干活的工作线程，干什么活呢？就是执行 SocketProcessor 的 run 方法，也就是解析请求并通过容器来处理请求，最终会调用到我们的 Servlet。后面我会用专门的篇幅介绍 Tomcat 怎么扩展和使用 Java 原生的线程池。

## 高并发思路

在弄清楚 NioEndpoint 的实现原理后，我们来考虑一个重要的问题，怎么把这个过程做到高并发呢？

高并发就是能快速地处理大量的请求，需要合理设计线程模型让 CPU 忙起来，尽量不要让线程阻塞，因为一阻塞，CPU 就闲下来了。另外就是有多少任务，就用相应规模的线程数去处理。我们注意到 NioEndpoint 要完成三件事情：接收连接、检测 I/O 事件以及处理请求，那么最核心的就是把这三件事情分开，用不同规模的线程去处理，比如用专门的线程组去跑 Acceptor，并且 Acceptor 的个数可以配置；用专门的线程组去跑 Poller，Poller 的个数也可以配置；最后具体任务的执行也由专门的线程池来处理，也可以配置线程池的大小。

## 本期精华

I/O 模型是为了解决内存和外部设备速度差异的问题。我们平时说的**阻塞或非阻塞**是指应用程序在**发起 I/O 操作时，是立即返回还是等待**。而**同步和异步**，是指应用程序在与内核通信时，**数据从内核空间到应用空间的拷贝，是由内核主动发起还是由应用程序来触发**。

在 Tomcat 中，EndPoint 组件的主要工作就是处理 I/O，而 NioEndpoint 利用 Java NIO API 实现了多路复用 I/O 模型。其中关键的一点是，读写数据的线程自己不会阻塞在 I/O 等待上，而是把这个工作交给 Selector。同时 Tomcat 在这个过程中运用到了很多 Java 并发编程技术，比如 AQS、原子类、并发容器，线程池等，都值得我们去细细品味。

## 课后思考

Tomcat 的 NioEndpoint 组件的名字中有 NIO，NIO 是非阻塞的意思，似乎说的是同步非阻塞 I/O 模型，但是 NioEndpoint 又是调用 Java 的 Selector 来实现的，我们知道 Selector 指的是 I/O 多路复用器，也就是我们说的 I/O 多路复用模型，这不是矛盾了吗？

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。

上一篇 13 | 热点问题答疑 ( 1 ) : 如何学习源码 ?

下一篇 15 | Nio2Endpoint组件 : Tomcat如何实现异步I/O ?

## 精选留言 (42)

写留言



2019-06-11

4

Io多路复用实际上也是同步非阻塞模式，用户线程阻塞在selector方法上，不像其他Io阻塞在read write 方法调用。同步是指当内核准备好数据时，还是应用程序线程把内核数据同步到用户空间。



QQ怪

2019-06-12

3

当老师谈到阻塞和非阻塞及异步和同步概念的时候，我对比了很多其他网上的解释，发现我不太理解他们的，但唯独老师总结的真的很很透彻、很到位！！！崇拜老师，发现这个专栏真的很棒！！！！



Monday

2019-06-12

2

### 阻塞与同异步的区别

本节的总结有如下的2句话，1) 阻塞与非阻塞指的是应用程序发起i/o操作后是等待还是立即返回。2) 同步与异步指的是应用程序在与内核通信时，数据从内核空间到应用空间的拷贝内核主动发起还是应用程序触发。

1，阻塞对应的是等待，非阻塞对应的是立即返回。这句应该好理解。...

展开

作者回复: 同步异步可以理解为谁主动，同步就是A问B要东西，总是A主动“伸手”问B要。异步就是A向B注册一个需求，货到了B主动“伸手”把货交给A。

阻塞队列在阻塞一个线程时，会有系统调用，有系统调用内核就要参与，只是这里的阻塞跟IO的阻塞是两回事。

其实不要迷茫，理解上面那几张图就行了。☺



Geek\_28b75...

2019-06-11

👍 2

问一个基础问题，线程的同步，和本节所讲的同步，意义上的不同

作者回复: 不是一个概念，线程的同步一般指对共享变量的互斥访问。IO模型的同步是指应用和内核的交互方式。



2019-06-11

👍 2

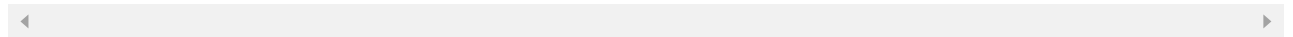
老师，操作系统级的连接指的是什么啊？

作者回复: TCP三次握手建立连接的过程中，内核通常会为每一个LISTEN状态的Socket维护两个队列：

SYN队列（半连接队列）：这些连接已经接到客户端SYN；

ACCEPT队列（全连接队列）：这些连接已经接到客户端的ACK，完成了三次握手，等待被accept系统调用取走。

Acceptor负责从ACCEPT队列中取出连接，当Acceptor处理不过来时，连接就堆积在ACCEPT队列中，这个队列长度也可以通过参数设置。



西兹兹

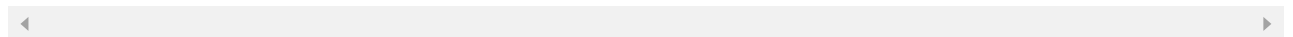
2019-06-15

👍 1

请问poller线程数是sever xml那个参数配置？暂时想到的是acceptorThreadCount对应acceptor线程数和acceptCount是待连接队列，那poller线程数对应哪个参数？

展开 ▾

作者回复: pollerThreadCount



WL

2019-06-13

👍 1

有一个问题请教一下老师：NioEndpoint中的events和eventCache之间是啥关系，在

events()方法中时把events中的元素放到eventCache中，而在add(final NioChannel socket, final int interestOps)方法中有吧events中的元素取出来放到eventCache中，我看了半天也不太理解这样实现的目的是啥，请老师指点一下。

展开 ∨



QQ怪

2019-06-13

👍 1

老师，信号驱动式 I/O与其他io模型的有啥不一样？

作者回复: 可以把信号驱动io理解为“半异步”，非阻塞模式是应用不断发起read调用查询数据到了内核没有，而信号驱动把这个过程异步化了，应用发起read调用时注册了一个信号处理函数，其实是个回调函数，数据到了内核后，内核触发这个回调函数，应用在回调函数里再发起一次read调用去读内核的数据。所以是半异步。

◀

▶



永恒记忆

2019-06-12

👍 1

老师，请教下，“当客户端发起一个http请求时，首先由Acceptor线程run方法中的socket = endpoint.serverSocketAccept();接收连接，然后传递给名称为Poller的线程去侦测I/O事件，Poller线程会一直select，选出内核将数据从网卡拷贝到内核空间的channel（也就是内核已经准备好数据）然后交给名称为Catalina-exec的线程去处理，这个过程也包括内核将数据从内核空间拷贝到用户空间这么一个过程，所以对于exec线程...

展开 ∨

作者回复: 1，理解的很准确

2，对的

3，Selector发出的select调用就是一个I/O操作。

◀

▶



QQ怪

2019-06-12

👍 1

老师我读完这篇文章我瞬间有点顿悟，真的很棒！！！加油！支持老师！



刘章周

2019-06-12

👍 1



同时有多个 Poller 线程在运行，每个 Poller 线程都有自己的 Queue。每个 Poller 线程可能同时被多个 Acceptor 线程调用来注册 PollerEvent。

老师:按照这个意思，有可能一个channel被多个selector监听，这样的话，重复监听的channel,造成资源浪费。

展开 ∨

作者回复: Acceptor接收到一个新的连接 ( channel ) 只注册到一个poller上，只是下一个新连接会注册到另一个poller上



二两豆腐

2019-06-11

1

老师，在“准备数据”阶段，这个阶段数据到底在准备的是什麼，“数据就绪”，指的是一种什么样的状态，什么样的数据才算是就绪了啊。

展开 ∨

作者回复: 浏览器的请求数据包到达网卡后，网卡通过硬件中断数据包拷贝到内核空间，然后内核做TCP/IP层的解包和重组，还原成完整的请求数据，这个时候算数据就绪了。



z.l

2019-06-15



```
serverSock = ServerSocketChannel.open();
serverSock.socket().bind(addr,getAcceptCount());
serverSock.configureBlocking(true);
```

请假下，这里为什么设置成true了？设置成true和false的区别是什么？

展开 ∨

作者回复: true表明服务端的监听socket是阻塞的，accept调用会阻塞



Geek\_28b75...

2019-06-14



selector接收acceptor传递的已连接的管道，那么selector岂不是只检测读事件就行了？acceptor已经完成了连接进入事件

展开 ▾

作者回复: 对的

◀ ▶



**Geek\_28b75...**

2019-06-14



老师，springboot应用程序之所以启动之后没有立即结束，本质原因就在于tomcat启动后，socket的accept（）方法阻塞监听吧？再加上此方法在死循环内部，用户线程不死

展开 ▾

作者回复: 不是的，Tomcat会停在StandardServer的await()方法

◀ ▶



**Francis**

2019-06-13



我理解，阻塞非阻塞说的是请求响应的环节，请求等待响应，那就是阻塞了；同步异步说的是数据转移的环节，从内核态准备好到同步到用户态，这个需要等待的环节这段是同步的。而多路复用是io复用的一种策略，细分select、poll、epoll等。



**星火燎原**

2019-06-13



阻塞 用户线程会一直在那里等待数据，  
非阻塞 用户线程不会等待，而是在轮询数据有没有到。  
老师我这样理解有问题吗？

作者回复: 对的

◀ ▶



**🌸🌸🌸...**

2019-06-13



老师，今天又看了一遍NioEndpoint的源码，发现PollerEvent实现了Runnable接口，但是还是直接调用run（）方法没有起线程调用。在events（）函数里。既然不用线程还实现Runnable接口，感觉误导读者这是现场运行的



J

2019-06-13



老师，你好，问一个与本节无关的内容。

最近学习socket编程，自己简单实现了一个ServerSocket监听本地端口，然后通过浏览器访问，尝试在代码里通过socket获取输入流解析http报文的时候，发现  
inputStream.read()一直读取不到-1，然后就无法走下一步了，不知道tomcat是如何解析http报文的呢？...

展开 ∨



永恒记忆

2019-06-13



老师好，想问下，刚刚debug发现只发一个请求，在Acceptor的run方法里面为什么跑了2遍的 `socket = endpoint.serverSocketAccept();` 不是应该没有新连接就阻塞在这里，有新连接就唤醒，然后处理后继续阻塞，可以第二遍循环还是获取到了socket，等第三遍循环就阻塞住了，一个请求为什么有2个新连接呢？

展开 ∨