

第11讲 | Java提供了哪些IO方式？NIO如何实现多路复用？

2018-05-29 杨晓峰

Java核心技术36讲

[进入课程 >](#)



讲述：黄洲君

时长 11:41 大小 5.35M



IO 一直是软件开发中的核心部分之一，伴随着海量数据增长和分布式系统的发展，IO 扩展能力愈发重要。幸运的是，Java 平台 IO 机制经过不断完善，虽然在某些方面仍有不足，但已经在实践中证明了其构建高扩展性应用的能力。

今天我要问你的问题是，Java 提供了哪些 IO 方式？NIO 如何实现多路复用？

典型回答

Java IO 方式有很多种，基于不同的 IO 抽象模型和交互方式，可以进行简单区分。

首先，传统的 java.io 包，它基于流模型实现，提供了我们最熟知的一些 IO 功能，比如 File 抽象、输入输出流等。交互方式是同步、阻塞的方式，也就是说，在读取输入流或者写

入输出流时，在读、写动作完成之前，线程会一直阻塞在那里，它们之间的调用是可靠的线性顺序。

java.io 包的好处是代码比较简单、直观，缺点则是 IO 效率和扩展性存在局限性，容易成为应用性能的瓶颈。

很多时候，人们也把 java.net 下面提供的部分网络 API，比如 Socket、ServerSocket、HttpURLConnection 也归类到同步阻塞 IO 类库，因为网络通信同样是 IO 行为。

第二，在 Java 1.4 中引入了 NIO 框架（java.nio 包），提供了 Channel、Selector、Buffer 等新的抽象，可以构建多路复用的、同步非阻塞 IO 程序，同时提供了更接近操作系统底层的高性能数据操作方式。

第三，在 Java 7 中，NIO 有了进一步的改进，也就是 NIO 2，引入了异步非阻塞 IO 方式，也有很多人叫它 AIO（Asynchronous IO）。异步 IO 操作基于事件和回调机制，可以简单理解为，应用操作直接返回，而不会阻塞在那里，当后台处理完成，操作系统会通知相应线程进行后续工作。

考点分析

我上面列出的回答是基于一种常见分类方式，即所谓的 BIO、NIO、NIO 2（AIO）。

在实际面试中，从传统 IO 到 NIO、NIO 2，其中有很多地方可以扩展开来，考察点涉及方方面面，比如：

基础 API 功能与设计，InputStream/OutputStream 和 Reader/Writer 的关系和区别。

NIO、NIO 2 的基本组成。

给定场景，分别用不同模型实现，分析 BIO、NIO 等模式的设计和实现原理。

NIO 提供的高性能数据操作方式是基于什么原理，如何使用？

或者，从开发者的角度来看，你觉得 NIO 自身实现存在哪些问题？有什么改进的想法吗？

IO 的内容比较多，专栏一讲很难能够说清楚。IO 不仅仅是多路复用，NIO 2 也不仅仅是异步 IO，尤其是数据操作部分，会在专栏下一讲详细分析。

知识扩展

首先，需要澄清一些基本概念：

区分同步或异步（synchronous/asynchronous）。简单来说，同步是一种可靠的有序运行机制，当我们进行同步操作时，后续的任务是等待当前调用返回，才会进行下一步；而异步则相反，其他任务不需要等待当前调用返回，通常依靠事件、回调等机制来实现任务间次序关系。

区分阻塞与非阻塞（blocking/non-blocking）。在进行阻塞操作时，当前线程会处于阻塞状态，无法从事其他任务，只有当条件就绪才能继续，比如 ServerSocket 新连接建立完毕，或数据读取、写入操作完成；而非阻塞则是不管 IO 操作是否结束，直接返回，相应操作在后台继续处理。

不能一概而论认为同步或阻塞就是低效，具体还要看应用和系统特征。

对于 [java.io](#)，我们都非常熟悉，我这里就从总体上进行一下总结，如果需要学习更加具体的操作，你可以通过[教程](#)等途径完成。总体上，我认为你至少需要理解：

IO 不仅仅是对文件的操作，网络编程中，比如 Socket 通信，都是典型的 IO 操作目标。

输入流、输出流（InputStream/OutputStream）是用于读取或写入字节的，例如操作图片文件。

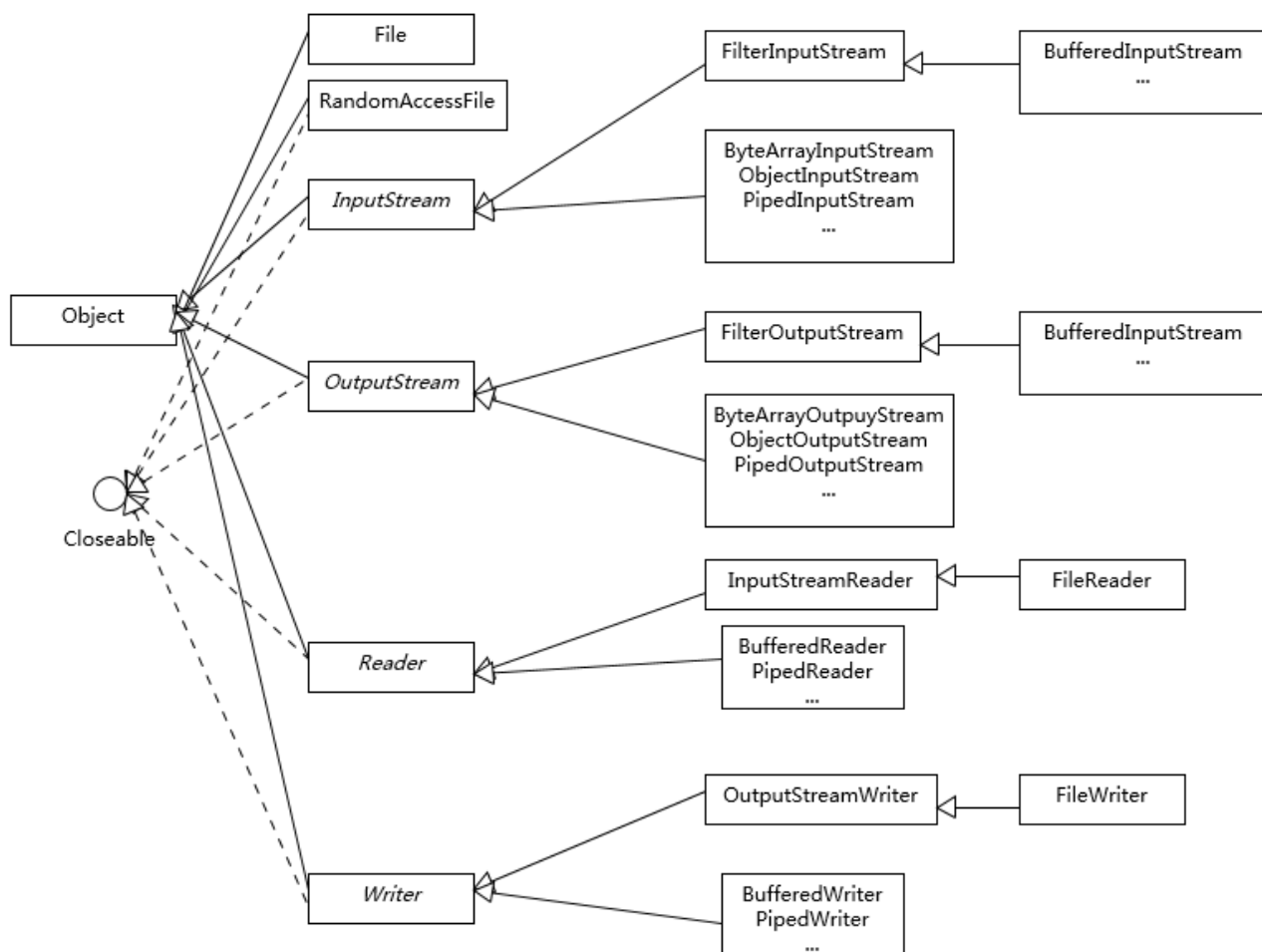
而 Reader/Writer 则是用于操作字符，增加了字符编解码等功能，适用于类似从文件中读取或者写入文本信息。本质上计算机操作的都是字节，不管是网络通信还是文件读取，Reader/Writer 相当于构建了应用逻辑和原始数据之间的桥梁。

BufferedOutputStream 等带缓冲区的实现，可以避免频繁的磁盘读写，进而提高 IO 处理效率。这种设计利用了缓冲区，将批量数据进行一次操作，但在使用中千万别忘了 flush。

参考下面这张类图，很多 IO 工具类都实现了 Closeable 接口，因为需要进行资源的释放。比如，打开 FileInputStream，它就会获取相应的文件描述符（FileDescriptor），需要利用 try-with-resources、try-finally 等机制保证 FileInputStream 被明确关闭，

进而相应文件描述符也会失效，否则将导致资源无法被释放。利用专栏前面的内容提到的 Cleaner 或 finalize 机制作为资源释放的最后把关，也是必要的。

下面是我整理的一个简化版的类图，阐述了日常开发应用较多的类型和结构关系。



1. Java NIO 概览

首先，熟悉一下 NIO 的主要组成部分：

Buffer，高效的数据容器，除了布尔类型，所有原始数据类型都有相应的 Buffer 实现。

Channel，类似在 Linux 之类操作系统上看到的文件描述符，是 NIO 中被用来支持批量式 IO 操作的一种抽象。


File 或者 Socket，通常被认为是比较高层次的抽象，而 Channel 则是更加操作系统底层的一种抽象，这也使得 NIO 得以充分利用现代操作系统底层机制，获得特定场景的性能优化，例如，DMA (Direct Memory Access) 等。不同层次的抽象是相互关联的，我们可以通过 Socket 获取 Channel，反之亦然。

Selector，是 NIO 实现多路复用的基础，它提供了一种高效的机制，可以检测到注册在 Selector 上的多个 Channel 中，是否有 Channel 处于就绪状态，进而实现了单线程对多 Channel 的高效管理。

Selector 同样是基于底层操作系统机制，不同模式、不同版本都存在区别，例如，在最新的代码库里，相关实现如下：


Linux 上依赖于

epoll (<http://hg.openjdk.java.net/jdk/jdk/file/d8327f838b88/src/java.base/linux/classes/sun/nio/ch/EPollSelectorImpl.java>) 。

 复制代码

```
1 Windows 上 NIO2 (AIO) 模式则是依赖于 iocp (http://hg.openjdk.java.net/jdk/jdk/file/d8327f838b88/src/java.base/windows/classes/sun/nio/ch/WindowsSelectorImpl.java) 。
```

Charset，提供 Unicode 字符串定义，NIO 也提供了相应的编解码器等，例如，通过下面的方式进行字符串到 ByteBuffer 的转换：


 复制代码

```
1 Charset.defaultCharset().encode("Hello world!");
```

2.NIO 能解决什么问题？

下面我通过一个典型场景，来分析为什么需要 NIO，为什么需要多路复用。设想，我们需要实现一个服务器应用，只简单要求能够同时服务多个客户端请求即可。

使用 java.io 和 java.net 中的同步、阻塞式 API，可以简单实现。

 复制代码

```
1 public class DemoServer extends Thread {
2     private ServerSocket serverSocket;
3     public int getPort() {
4         return serverSocket.getLocalPort();
5     }
6     public void run() {
```

```

7      try {
8          serverSocket = new ServerSocket(0);
9          while (true) {
10             Socket socket = serverSocket.accept();
11             RequestHandler requestHandler = new RequestHandler(socket);
12             requestHandler.start();
13         }
14     } catch (IOException e) {
15         e.printStackTrace();
16     } finally {
17         if (serverSocket != null) {
18             try {
19                 serverSocket.close();
20             } catch (IOException e) {
21                 e.printStackTrace();
22             }
23         }
24     }
25 }
26
27 public static void main(String[] args) throws IOException {
28     DemoServer server = new DemoServer();
29     server.start();
30     try (Socket client = new Socket(InetAddress.getLocalHost(), server.getPort())) {
31         BufferedReader bufferedReader = new BufferedReader(new InputStreamReader(client.getInputStream()));
32         bufferedReader.lines().forEach(s -> System.out.println(s));
33     }
34 }
35
36 // 简化实现，不做读取，直接发送字符串
37 class RequestHandler extends Thread {
38     private Socket socket;
39     RequestHandler(Socket socket) {
40         this.socket = socket;
41     }
42     @Override
43     public void run() {
44         try (PrintWriter out = new PrintWriter(socket.getOutputStream())) {
45             out.println("Hello world!");
46             out.flush();
47         } catch (Exception e) {
48             e.printStackTrace();
49         }
50     }
51 }
52

```

其实现要点是：

服务器端启动 `ServerSocket`，端口 0 表示自动绑定一个空闲端口。

调用 `accept` 方法，阻塞等待客户端连接。

利用 `Socket` 模拟了一个简单的客户端，只进行连接、读取、打印。


当连接建立后，启动一个单独线程负责回复客户端请求。

这样，一个简单的 `Socket` 服务器就被实现出来了。

思考一下，这个解决方案在扩展性方面，可能存在什么潜在问题呢？

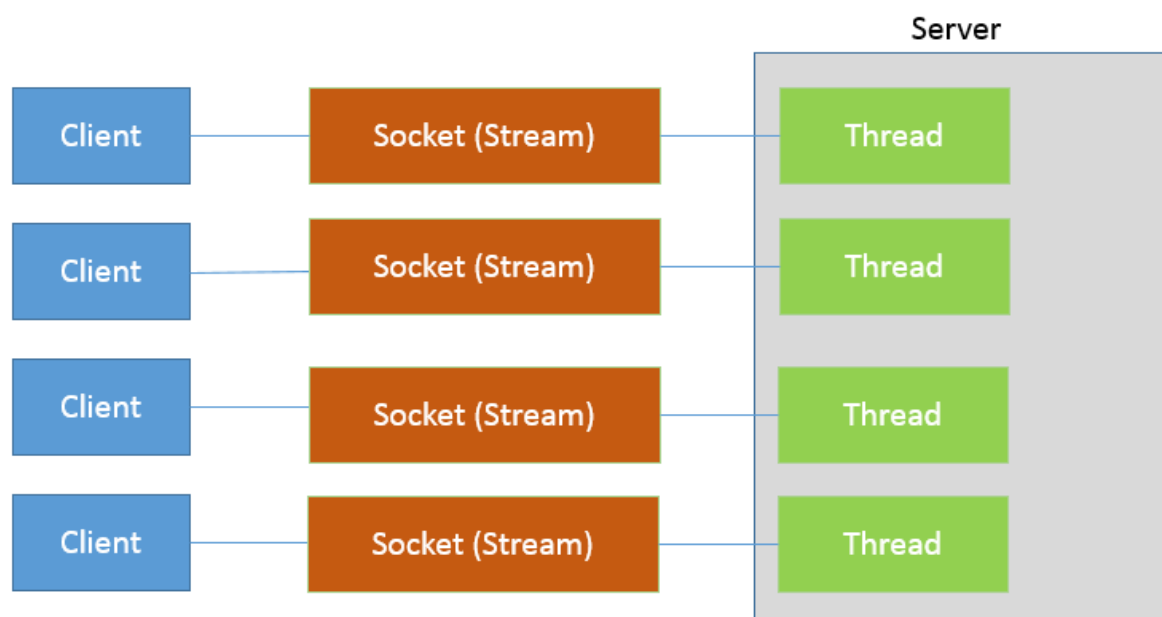
大家知道 Java 语言目前的线程实现是比较重量级的，启动或者销毁一个线程是有明显开销的，每个线程都有单独的线程栈等结构，需要占用非常明显的内存，所以，每一个 `Client` 启动一个线程似乎都有些浪费。

那么，稍微修正一下这个问题，我们引入线程池机制来避免浪费。

 复制代码


```
1 serverSocket = new ServerSocket(0);
2 executor = Executors.newFixedThreadPool(8);
3 while (true) {
4     Socket socket = serverSocket.accept();
5     RequestHandler requestHandler = new RequestHandler(socket);
6     executor.execute(requestHandler);
7 }
8
```

这样做似乎好了很多，通过一个固定大小的线程池，来负责管理工作线程，避免频繁创建、销毁线程的开销，这是我们构建并发服务的典型方式。这种工作方式，可以参考下图来理解。



如果连接数并不是非常多，只有最多几百个连接的普通应用，这种模式往往可以工作的很好。但是，如果连接数量急剧上升，这种实现方式就无法很好地工作了，因为线程上下文切换开销会在高并发时变得很明显，这是同步阻塞方式的低扩展性劣势。

NIO 引入的多路复用机制，提供了另外一种思路，请参考我下面提供的新的版本。

 复制代码

```
1 public class NIOServer extends Thread {
2     public void run() {
3         try (Selector selector = Selector.open();
4             ServerSocketChannel serverSocket = ServerSocketChannel.open();) { // 创建 Se
5             serverSocket.bind(new InetSocketAddress(InetAddress.getLocalHost(), 8888));
6             serverSocket.configureBlocking(false);
7             // 注册到 Selector，并说明关注点
8             serverSocket.register(selector, SelectionKey.OP_ACCEPT);
9             while (true) {
10                 selector.select(); // 阻塞等待就绪的 Channel，这是关键点之一
11                 Set<SelectionKey> selectedKeys = selector.selectedKeys();
12                 Iterator<SelectionKey> iter = selectedKeys.iterator();
13                 while (iter.hasNext()) {
14                     SelectionKey key = iter.next();
15                     // 生产系统中一般会额外进行就绪状态检查
16                     sayHelloWorld((ServerSocketChannel) key.channel());
17                     iter.remove();
18                 }
19             }
20         }
21     }
22 }
```



```
20         } catch (IOException e) {
21             e.printStackTrace();
22         }
23     }
24     private void sayHelloWorld(ServerSocketChannel server) throws IOException {
25         try (SocketChannel client = server.accept();) {
26             client.write(Charset.defaultCharset().encode("Hello World"));
27         }
28     }
29     // 省略了与前面类似的 main
30 }
```

这个非常精简的样例掀开了 NIO 多路复用的面纱，我们可以分析下主要步骤和元素：

首先，通过 `Selector.open()` 创建一个 `Selector`，作为类似调度员的角色。

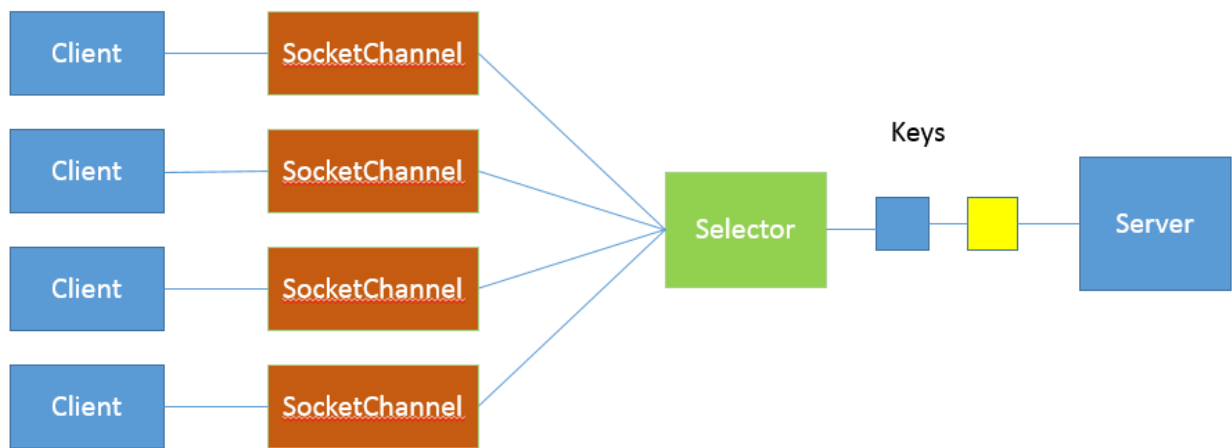
然后，创建一个 `ServerSocketChannel`，并且向 `Selector` 注册，通过指定 `SelectionKey.OP_ACCEPT`，告诉调度员，它关注的是新的连接请求。

注意，为什么我们要明确配置非阻塞模式呢？这是因为阻塞模式下，注册操作是不允许的，会抛出 `IllegalBlockingModeException` 异常。

`Selector` 阻塞在 `select` 操作，当有 `Channel` 发生接入请求，就会被唤醒。

在 `sayHelloWorld` 方法中，通过 `SocketChannel` 和 `Buffer` 进行数据操作，在本例中是发送了一段字符串。

可以看到，在前面两个样例中，IO 都是同步阻塞模式，所以需要多线程以实现多任务处理。而 NIO 则是利用了单线程轮询事件的机制，通过高效地定位就绪的 `Channel`，来决定做什么，仅仅 `select` 阶段是阻塞的，可以有效避免大量客户端连接时，频繁线程切换带来的问题，应用的扩展能力有了非常大的提高。下面这张图对这种实现思路进行了形象地说明。



在 Java 7 引入的 NIO 2 中，又增添了一种额外的异步 IO 模式，利用事件和回调，处理 Accept、Read 等操作。AIO 实现看起来是类似这样子：

[复制代码](#)

```
1 AsynchronousServerSocketChannel serverSock = AsynchronousServerSocketChannel.open();
2 serverSock.accept(serverSock, new CompletionHandler<>() { // 为异步操作指定 CompletionHandler
3     @Override
4     public void completed(AsynchronousSocketChannel sockChannel, AsynchronousServerSocketChannel serverSock) {
5         serverSock.accept(serverSock, this);
6         // 另外一个 write(sockChannel, CompletionHandler{})
7         sayHelloWorld(sockChannel, Charset.defaultCharset().encode("Hello World!"));
8     }
9 });
10 // 省略其他路径处理方法...
11 });
```

鉴于其编程要素（如 Future、CompletionHandler 等），我们还没有进行准备工作，为避免理解困难，我会在专栏后面相关概念补充后再进行介绍，尤其是 Reactor、Proactor 模式等方面将在 Netty 主题一起分析，这里我先进行概念性的对比：

基本抽象很相似，AsynchronousServerSocketChannel 对应于上面例子中的 ServerSocketChannel；AsynchronousSocketChannel 则对应 SocketChannel。

业务逻辑的关键在于，通过指定 CompletionHandler 回调接口，在 accept/read/write 等关键节点，通过事件机制调用，这是非常不同的一种编程思路。

今天我初步对 Java 提供的 IO 机制进行了介绍，概要地分析了传统同步 IO 和 NIO 的主要组成，并根据典型场景，通过不同的 IO 模式进行了实现与拆解。专栏下一讲，我还将继续分析 Java IO 的主题。

一课一练

关于今天我们讨论的题目你做到心中有数了吗？留一道思考题给你，NIO 多路复用的局限性是什么呢？你遇到过相关的问题吗？

请你在留言区写写你对这个问题的思考，我会选出经过认真思考的留言，送给你一份学习鼓励金，欢迎你与我一起讨论。

你的朋友是不是也在准备面试呢？你可以“请朋友读”，把今天的题目分享给好友，或许你能帮到他。

 极客时间

Java 核心技术 36 讲

—— 前 Oracle 首席工程师
带你修炼 Java 内功 ——

杨晓峰 前 Oracle 首席工程师



新版升级：点击「 请朋友读」，10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 第10讲 | 如何保证集合是线程安全的？ConcurrentHashMap如何实现高效地线程安全？

下一篇 第12讲 | Java有几种文件拷贝方式？哪一种最高效？

精选留言 (49)

写留言



I am a ...

2018-05-29

64

由于nio实际上是同步非阻塞io，是一个线程在同步的进行事件处理，当一组事channel处理完毕以后，去检查有没有又可以处理的channel。这也就是同步+非阻塞。同步，指每个准备好的channel处理是依次进行的，非阻塞，是指线程不会傻傻的等待读。只有当channel准备好后，才会进行。那么就会有这样一个问题，当每个channel所进行的都是耗时操作时，由于是同步操作，就会积压很多channel任务，从而完成影响。那么就需要对...
展开



王睿

2018-09-13

51

举个收快递的例子不知道理解是否正确。

BIO，快递员通知你有一份快递会在今天送到某某地方，你需要在某某地方一致等待快递员的到来。

...

展开



明翼

2018-07-05

29

看完之后还是不了解nio，感觉看起来越来越吃力了，大半天都啃不了一篇，好多东西都不熟悉，还要自己查资料去了解然后再回过来看，，，老师最好给些学习的资料让我们能找到，就这一篇感觉根本不够



Chan

2018-06-16

29

B和N通常是针对数据是否就绪的处理方式来
sync和async是对阻塞进行更深一层次的阐释，区别在于数据拷贝由用户线程完成还是内核完成，讨论范围一定是两个线程及以上了。

...

展开



雷霹雳的爸...

2018-05-29

👍 20

批评NIO确实要小心，我觉得主要是三方面，首先是如果是从写BIO过来的同学，需要有一个巨大的观念上的转变，要清楚网络就是并非时刻可读可写，我们用NIO就是在认真的面
对这个问题，别把channel当流往死里用，没读出来写不进去的时候，就是该考虑让度线程
资源了，第二点是NIO在不同的平台上的实现方式是不一样的，如果你工作用电脑是win，
生产是linux，那么建议直接在linux上调试和测试，第三点，概念上的，理解了会在各方...
展开 ▾

作者回复: 不错，不过，在非常有必要之前，不见得都要底层，毕竟各种抽象，都是为特定领域工
程师准备的，JMM等抽象都是为了大家有个清晰的、不同层面的高效交流



Allen

2018-06-30

👍 9

希望能听到更多原理性的东西，而不是在网上能搜到的样例代码



aiwen

2018-06-02

👍 9

到底啥是多路复用？一个线程管理多个链接就是多路复用？



loranceche...

2018-05-31

👍 7

我也自己写过一个基于nio2的网络程序，觉得配合futrue写起来很舒服。

仓库地址：<https://github.com/LoranceChen/RxSocket> 欢迎相互交流开发经验 ~

记得在netty中，有一个搁置的netty5.x项目被废弃掉了，原因有一点官方说是性能提升不
明显，这是可以理解的，因为linux下是基于epoll，本质还是select操作。...

展开 ▾

作者回复: 坦白说，内核epoll之类实现细节目前我的理解也有限



zjh

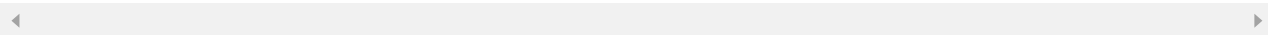
2018-05-31

👍 6

看nio代码部分，请求接受和处理都是一个线程在做。这样的话，如果有多个请求过来都是按顺序处理吧，其中一个处理时间比较耗时的话那所有请求不都卡住了吗？如果把nio的处理部分也改成多线程会有什么问题吗

展开 ▾

作者回复: 这种情况需要考虑把耗时操作并发处理，再说处理是费cpu，还是重io，需要不同处理；如果耗时操作非常多，就不符合这种模型的适用场景



逐梦之音

2018-05-29

👍 5

IO的调用可以分为三大块，请求调用，逻辑处理，响应返回处理。常规的BIO在这三个阶段会串行的阻塞的。NIO其实可以理解为将这三个阶段尽可能的去阻塞或者减少阻塞。看了上面的例子，NIO的服务器端在接受客户端请求的时候，是单线程执行的，而BIO是多线程处理的。但是不管咋的，他们服务器端处理具体的客户业务逻辑是都要用多线程的吧？

展开 ▾



Chan

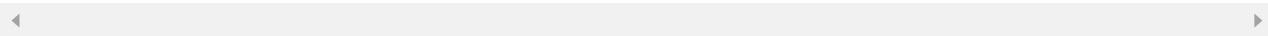
2018-06-16

👍 4

忘记回答问题了。所以对于多路复用IO，当出现有的IO请求在数据拷贝阶段，会出现由于资源类型过份庞大而导致线程长期阻塞，最后造成性能瓶颈的情况

展开 ▾

作者回复: 对



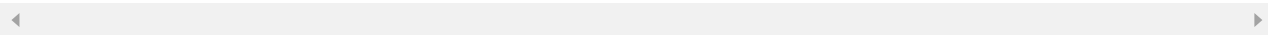
扁扁圆圆

2018-06-02

👍 4

这里Nio的Selector只注册了一个sever chanel，这没有实现多路复用吧，多路复用不是注册了多个channel，处理就绪的吗？而且处理客户端请求也是在同线程内，这还不如上面给的Bio解决方案吧

作者回复: 这是简化的例子，少占篇幅





ykkk88

2018-05-29

👍 4

这个nio看起来还是单线程在处理，如果放到多线程池中处理和bio加线程池有啥区别呢



扁担

2018-10-21

👍 3

据我理解，NIO2的局限性在于，如果回调时客户端做了重操作，就会影响调度，导致后续的client回调缓慢

作者回复: 对，这是这种多路复用的主要局限之一，nodejs等其他类似框架都有这问题



萧萧

2018-08-08

👍 3

作者对同步/异步，阻塞/非阻塞的概念说明存在问题。

《操作系统（第9版）》中关于进程通信中有对这部分概念做过解释，在进程间通信的维度，同步和阻塞，异步和非阻塞是相同的概念。

...

展开 ▾

作者回复: 这东西并没有完全共识，概念定义要看上下文，很多情况下可以算是同等，但在网络IO编程中是区分的，本文的关注点就是这个



张凯江

2018-07-18

👍 3

cpu运算密集型应用。node得诟病

展开 ▾



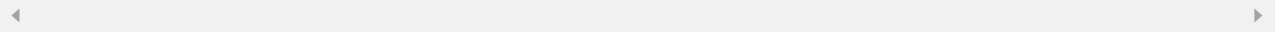
Miaoze

2018-06-05

👍 3

杨老师，把你给的NIOServer的例子做了一下，发现sayHelloWorld()方法，client.write()后，如果没有client.close(),线程一直在挂着。请确认一下，是否例子缺了client.close()？

作者回复: try with resource就相当于在finally里close；一直挂着是因为server在伺候



loranceche...

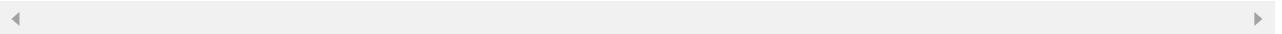
2018-05-31

👍 3

还有一个问题请教，select在单线程下处理监听任务是否会成为瓶颈？能否通过创建多个select实例，并发监听socket事件呢？

展开 ▾

作者回复: Doug Lea曾经推荐过多个Selector，也就是多个reactor，如果你是这意思



残月@诗雨

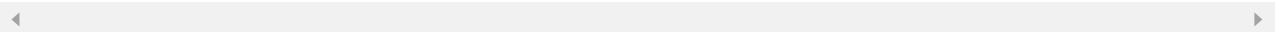
2018-05-29

👍 3

杨老师，有个问题一直不太明白：BufferedInputStream和普通的InputStream直接read到一个缓冲数组这两种方式有什么区别？

展开 ▾

作者回复: 我理解是bufferedIS是内部预读，所以两个buffer的意义不一样，前面是减少磁盘之类操作



L.B.Q.Y

2018-05-29

👍 3

NIO多路复用模式，如果对应事件的处理比较耗时，是不是会导致后续事件的响应出现延迟。

作者回复: 所以我理解，适用于大量请求大小有限的场景，（主任务）单线程模型，比如nodejs都有类似情况，

