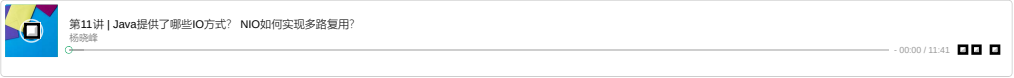
第11讲 | Java提供了哪些IO方式？ NIO如何实现多路复用？

2018-05-29 杨晓峰



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：

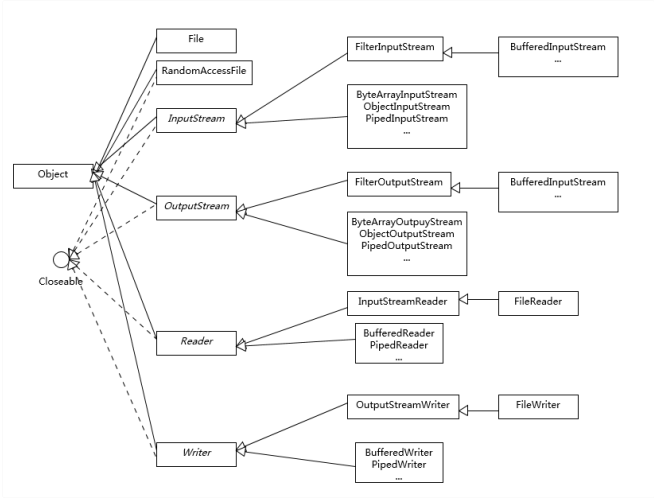
- 区分同步或异步（synchronous/asynchronous）。简单来说，同步是一种可靠的有序运行机制，当我们进行同步操作时，后续的任务是等待当前调用返回，才会进行下一步；而异步则相反，其他任务不需要等待当前调用返回，通常依靠事件、回调等机制来实现任务间次序关系。
- 区分阻塞与非阻塞（blocking/non-blocking）。在进行阻塞操作时，当前线程会处于阻塞状态，无法从事其他任务，只有当条件就绪才能继续，比如ServerSocket新连接建立完毕，或数据读取、写入操作完成；而非阻塞则是不管IO操作是否结束，直接返回，相应操作在后台继续处理。

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）。

  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/Iocp.java）。
- Chartset，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：

```
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。

```
public class DemoServer extends Thread {
    private ServerSocket serverSocket;
    public int getPort() {
        return  serverSocket.getLocalPort();
    }
    public void run() {
        try {
```

```
            serverSocket = new ServerSocket(0);
            while (true) {
                Socket socket = serverSocket.accept();
                RequestHandler requestHandler = new RequestHandler(socket);
                requestHandler.start();
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (serverSocket != null) {
                try {
                    serverSocket.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
                ;
            }
        }
    }
    public static void main(String[] args) throws IOException {
        DemoServer server = new DemoServer();
        server.start();
        try (Socket client = new Socket(InetAddress.getLocalHost(), server.getPort())) {
            BufferedReader bufferedReader = new BufferedReader(new                  InputStreamReader(client.getInputStream()));
            bufferedReader.lines().forEach(s -> System.out.println(s));
        }
    }
 }
// 简化实现，不做读取，直接发送字符串
class RequestHandler extends Thread {
    private Socket socket;
    RequestHandler(Socket socket) {
        this.socket = socket;
    }
    @Override
    public void run() {
        try (PrintWriter out = new PrintWriter(socket.getOutputStream());) {
            out.println("Hello world!");
            out.flush();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
 }
```

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。

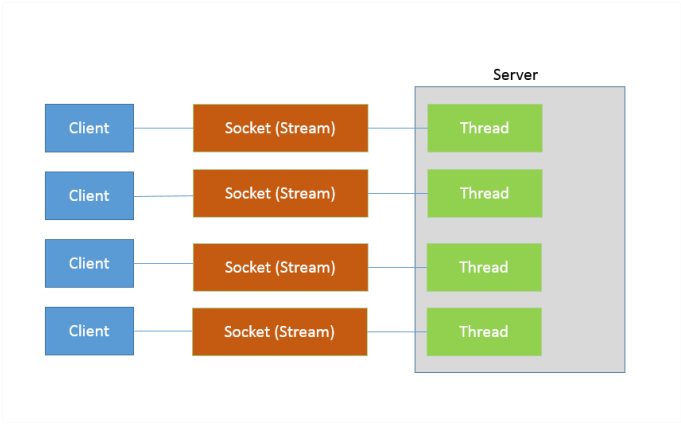
思考一下，这个解决方案在扩展性方面，可能存在什么潜在问题呢？

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。

```
serverSocket = new ServerSocket(0);
executor = Executors.newFixedThreadPool(8);
 while (true) {
    Socket socket = serverSocket.accept();
    RequestHandler requestHandler = new RequestHandler(socket);
    executor.execute(requestHandler);
}
```

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。

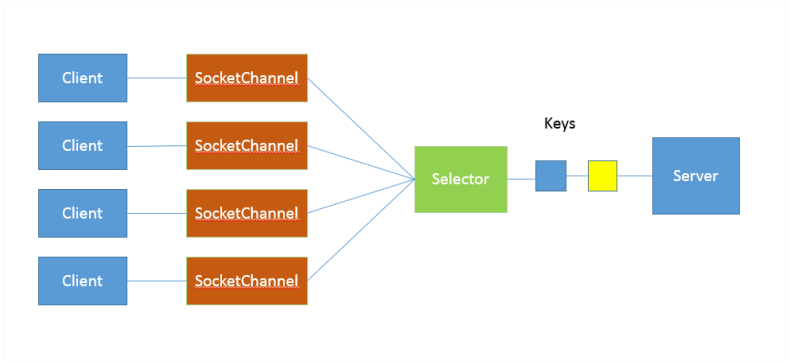
```
public class NIOServer extends Thread {
    public void run() {
        try (Selector selector = Selector.open();
             ServerSocketChannel serverSocket = ServerSocketChannel.open();) {// 创建Selector和Channel
            serverSocket.bind(new InetSocketAddress(InetAddress.getLocalHost(), 8888));
            serverSocket.configureBlocking(false);
            // 注册到Selector，并说明关注点
            serverSocket.register(selector, SelectionKey.OP_ACCEPT);
            while (true) {
                selector.select();// 阻塞等待就绪的Channel，这是关键点之一
                Set<SelectionKey> selectedKeys = selector.selectedKeys();
                Iterator<SelectionKey> iter = selectedKeys.iterator();
                while (iter.hasNext()) {
                    SelectionKey key = iter.next();
                   // 生产系统中一般会额外进行就绪状态检查
                    sayHelloWorld((ServerSocketChannel) key.channel());
                    iter.remove();
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    private void sayHelloWorld(ServerSocketChannel server) throws IOException {
        try (SocketChannel client = server.accept();) {         client.write(Charset.defaultCharset().encode("Hello world!"));
        }
    }
   // 省略了与前面类似的main
}
```

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：

```
AsynchronousServerSocketChannel serverSock =        AsynchronousServerSocketChannel.open().bind(sockAddr);
serverSock.accept(serverSock, new CompletionHandler<>() { //为异步操作指定CompletionHandler回调函数
    @Override
    public void completed(AsynchronousSocketChannel sockChannel, AsynchronousServerSocketChannel serverSock) {
        serverSock.accept(serverSock, this);
        // 另外一个 write（sock，CompletionHandler{}）
        sayHelloWorld(sockChannel, Charset.defaultCharset().encode
                ("Hello World!"));
    }
  // 省略其他路径处理方法...
});
```

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的再进行介绍，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。

I am a psycho

2018-05-29

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nio进行类似负载均衡的操作，如用线程池去进行管理读写，将channel分给其他的线程去执行，这样既充分利用了每一个线程，

又不至于都堆积在一个线程中，等待执行。杨老师，不知道上述理解是否正确？

雷霹雳的爸爸

2018-05-29

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，理解了会在各方面都有益处，NIO在IO操作本身上还是阻塞的，也就是他还是同步IO，AIO读写行为的回调才是异步IO，而这个真正实现，还是看系统底层的，写完之后，我觉得我这一二三有点凑数的嫌疑

作者回复

2018-05-29

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

逐梦之音

2018-05-29

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？

灰飞灰猪不会灰飞.烟灭

2018-05-29

老师 注册管道到select上，应该用队列实现的吧？
开启一个线程大概需要多少内存开销呢，我记得数据库连接大概2M

作者回复

2018-05-29

线程看定义stack大小等，32、64位都不一样

Chan

2018-06-16

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。

同步阻塞，从数据是否准备就绪到数据拷贝都是由用户线程完成

同步非阻塞，数据是否准备就绪由内核判断，数据拷贝还是用户线程完成

异步非阻塞，数据是否准备就绪到数据拷贝都是内核来完成

所以真正的异步IO一定是非阻塞的。

多路复用IO即使有Reactor通知用户线程也是同步IO范畴，因为数据拷贝期间仍然是用户线程完成。

所以假如我们没有内核支持数据拷贝的情况下，讨论的非阻塞并不是彻底的非阻塞，也就没有引入sync和async讨论的必要了

不知道这样理解是否正确

zjh

2018-05-31

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

作者回复

2018-06-06

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

残月@诗雨

2018-05-29

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？

作者回复

2018-05-29

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

明翼

2018-07-05

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

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

作者回复

2018-06-16

对

Miaozhe

2018-06-05

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

2018-06-06

try with resource就相当于在finally里close；一直挂着是因为server在伺服

扁扁圆圆

2018-06-02

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

2018-06-03

这是简化的例子，少占篇幅

aiwen

2018-06-02

到底啥是多路复用？一个线程管理多个链接就是多路复用？

lorancechen

2018-05-31

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。

听了课程之后，有一点印象比较深刻，select模式是使用一个线程做监听，而bio每次来一个链接都要做线程切换，所以节省的时间在线程切换上，当然如果是c/c++实现，原理也是一样的。

想问一个一直困惑的问题，select内部如何实现的呢？
个人猜测：不考虑内核，应用层的区分，单纯从代码角度考虑，我猜测，当select开始工作时，有一个定时器，比如每10ms去检查一下网络缓冲区中是否有tcp的链接请求包，然后把这些包筛选出来，作为一个集合（即代码中的迭代器）填入java select类的一个集合成员中，然后唤醒select线程，做一个while遍历处理链接请求，这样一次线程调度就可以处理10ms内的所有链接。与bio比，节省的时间在线程上下文切换上。不知道这么理解对不对。
另外，也希望能出一个课程，按照上面这种理解底层的方式，讲讲select（因为我平常工作在linux机器，所以对select epoll比较感兴趣）如何处理read，write操作的。谢谢～

作者回复

2018-06-01

坦白说，内核epoll之类实现细节目前我的理解也有限

RoverYe

2018-05-30

nio不适合数据量太大交互的场景

ykkk88

2018-05-29

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

L.B.Q.Y

2018-05-29

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

2018-05-29

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，

Forrest

2018-05-29

使用线程池可以很好的解决线程复用，避免线程创建带来的开销，效果也很好，一个问题想请教下，当线程池无法满足需要时可以用什么方式解决？
作者回复

2018-05-29

没有看明白问题，抱歉

张凯江

2018-07-18

cpu运算密集型应用。node得诟病

清风

2018-07-03

我认为这个nio 2基于事件编程就跟swing 编程添加监听事件一样，有事件发生了，执行回调函数。不知道理解是否准确。同时针对nio的采用线程不断地轮询，对客户多量大后会有响应延迟，并发数量有限。同时也想知道tomcat是怎么样通过nio 来解决着问题的。一直没有时间看一下他的源码，枉费我工作这么多年。老师可以写一篇这个nio和tomcat的文章！

Allen

2018-06-30

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

---

wenxueliu

2018-06-27

哈哈，如果能结合操作系统之中io模型讲也许更好，顺便分析下select和poll和epoll的实现机制，那么就比较完美了。不过话说这样，就这个主题就可以当一门课啦。推荐netty权威指南

Miaozhe

2018-06-06

NIO 2是借助AsychronousChannelGroup,实现多Channel方案，有具备异步功能，解决NIO 1单线程多通路问题。
不过异步有两种方案：
1.New一个CompletionHandler对象，在重写方法中处理，Write和Read.
2.通过concurrent.Future对象，建模一个挂起操作，之后可以通过Get获取socketChannel处理Write和Read。
想问问杨老师，两种方式各有什么优劣？

Miaozhe

2018-06-05

问个问题，看到你写的样例中，几处try (),小括号里面写了一些逻辑，这种写法跟放在{}里面有啥区别？望专家回复一下。
作者回复

2018-06-06

前面介绍过，try-with-resources，自动close

yxw

2018-06-03

java nio的selector主要的问题是效率，当并发连接数达到数万甚至数十万的时候，单线程的selector会是一个瓶颈；另一个问题就是再线上运行过程中经常出现cpu占用100%的情况，原因也是由于selector依赖的操作系统底层机制bug 导致的selector假死，需要程序重建selector来解决，这个问题再jdk中似乎并没有很好的解决，netty成为了线上更加可靠的网络框架。不知理解的是否正确，请老师指教。
作者回复

2018-06-05

嗯，有局限性；那个epoll的bug应该在8里修了，netty的改进不止那些，它为了性能改了很多底层，后面会介绍，好多算是hack；另外nio的目的是通用场景的基础API，和终端应用有个距离，核心类库很多都是如此定位，netty这种开源框架更贴近用户场景

lorancechen

2018-05-31

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？
作者回复

2018-05-31

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

Jerry银银

2018-05-30

我是来找茬的��：file 应该的读「fail」

请教一个概念性的问题：线程池模式和数据库连接池模式，是不是就是大家通常所说的对象池模式？

funnyx

2018-05-29

这个模式和go中的goroutine，channel，select很像，值得研究。

极客时间

极客时间