基于阻塞与非阻塞 I/O 网络模型的 Java 语言实现①

Implementation of Network Communication Mode Based on Java Blocking and Non – Blocking IO

袁劲松　马旭东　（东南大学 自动控制学院 江苏 南京 210096）

摘　要：本文对网络应用中阻塞通信与非阻塞通信工作机制及实现等问题进行了研究和探讨、提出了系统地实现阻塞与非阻塞通信的方法和步骤，文中对比了两种不同的网络通信方式，分别给出了基于阻塞与非阻塞 IO 开发高性能网络应用程序的具体实例。

关键词：NIO IO Java 阻塞 非阻塞 网络通信

1 引言

Java 平台传统的 I/O 系统基于 byte（字节）和 Stream（数据流），以字节为单位，以流的方式处理数据。而 NIO（New I/O）以块的方式处理数据，它的系统操作面向 Buffer（缓冲），Channel（通道）和 Selector（选择器），这种模式利用了操作系统管理内存和文件的方式，并将一些耗时操作直接转移给操作系统，使 I/O 的速度得以提高、性能得到了明显的改善。本文在研究和探讨 Java IO 和 NIO 特性及其阻塞和非阻塞通信工作机制的基础上，分别探索了实现阻塞和非阻塞通信的方法、步骤。并针对一个网络应用实例，给出了两种模式的 Java 实现。

2 阻塞式 I/O 工作原理

在 Java 平台上，处理有关 I/O 操作的方法通常是：当一个方法需要处理 I/O 有关的事务时，该方法立即被 Java 虚拟机设置成等待状态，直到有关的 I/O 操作完成。称这样的过程为阻塞 IO。基本上可以将阻塞式 I/O 的网络程序过程描述如下：

（1）打开一个 ServerSocket 监听端口。

（2）程序将阻塞，直到有连接请求。

（3）读取请求。

（4）发送响应。

（5）关闭连接。

（6）重复 2 – 5 步骤。

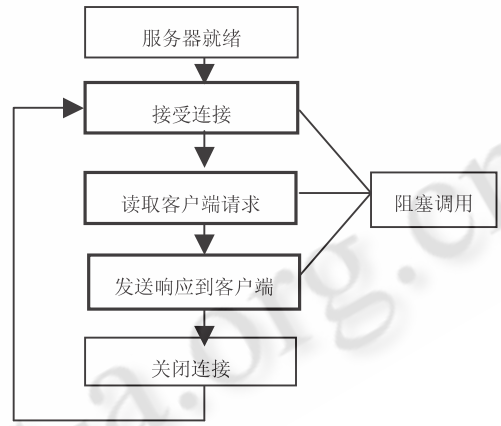

图 1　阻塞式 I/O 工作原理示意图

如图 1 所示。图中粗框的部分将被阻塞直至条件满足，这种模式导致过多的系统开销。通常我们设计多线程应用程序来解决这个问题（如图 2 所示），客户端向服务器提交申请，服务器生成一个拥有对应客户端的 socket 线程，每个客户端激活一个新的线程，该线程便成为服务代理，每个线程代理一个客户端，服务器继续侦听连接请求，如果某个客户端出现了问题，例如连接中断，并不会影响服务的运行，但是每个客户端都始终保持与相应线程的连接，线程之间是完全独立的。服务通过线程技术实现多链接，但是线程的建立需要较大的开销，数量较多的线程将会降低系统的运行速

① 基金项目：863 计划资助项目（2006AA040202，2007AA041703）

度。而且,我们并不是真正地需要如此大的开销来供应如此多的线程,这些线程没有有效地利用 CPU,它们将大多数时间花费在 I/O 的阻塞上,所以这些线程是低效率的。多线程服务器开发过程:(1)写服务器端程序。(2)在循环中调用多线程程序。(3)为多线程程序传递 Socket 参数。(4)写多线程程序。

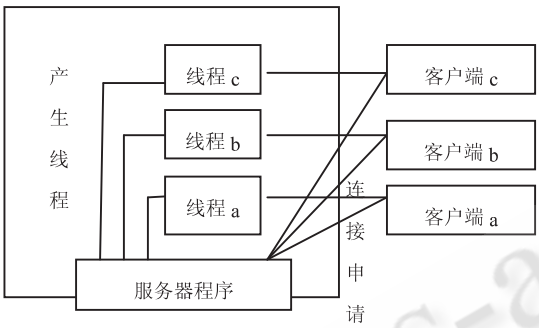

图 2　利用多线程解决阻塞示意图

3　非阻塞式 NI/O 工作原理

从 Java 1.4 开始,引入一组新的 API 来进行 I/O 操作(NIO API)。在 NIO 中,一个重要的特征就是可实现非阻塞的网络 I/O 操作(non – blocking I/O),可以编写出性能更好、更易扩展的网络应用程序,可以轻松的处理较多的连接,这些主要依靠新 I/O 引用的三个新概念和相应的工具包:

(1) 线性排列的数据可读写缓冲(Buffer)。

(2) 双向通道(Channel) – – 新的 I/O 抽象,在 NIO 中替代流。

(3) 选择器(Selectors)。

缓冲区(Buffer)实质上是一个容器对象,它包含一些要写入或者刚读出的数据。NIO 中加入缓冲对象,体现了新库与原 I/O 的一个重要区别。在面向流的 IO 中,是将数据直接写入或者将数据直接读到 Stream 对象中。而在 NIO 中,任何时候所有数据都是用缓冲区处理。

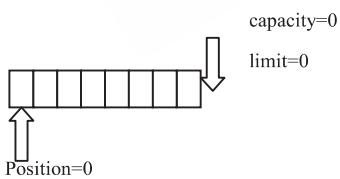

图 3　一个容量为 8 的 Buffer

通道(Channel)通常被认为代表了一个通向实体的连接,这个实体可以是文件、网络 Socket 或能够执行 IO 操作的程序组件。可以通过它读取和写入数据,它就像是流,但它是双向的,而流只是在一个方向上移动,在读写流时必须定义 InputStream 和 OutputStream 的对象,而通道可以用于读、写或者同时用于读写。NIO 中主要是利用通道(Channel)处理 Buffer 的数据。

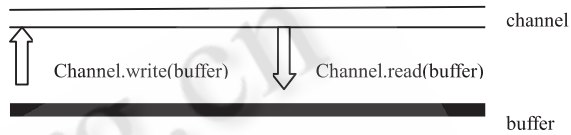

图 4　Channel 处理 Bufferr 的数据示意

选择器(Selectors)是 NIO 中重要的概念,如果我们使用新的 IO 构造非阻塞的网络服务,如何判断是否有请求发生呢? 这时我们就用的是 Selector 类,我们将一个可注册的 Channel 类 SelectableChannel 注册到一个 Selector 类中,Selector 将为我们监听所有的 I/0 操作。选择器通过一个符号(Selection Key)与被选择的通道交互,当一个通道被注册到选择器上的时候,选择器创建一个选择键与此通道相关联,在此后的操作中,选择器通过这个键操纵通道完成非阻塞的输入输出。

NIO 是一种没有阻塞地读写数据的方法。阻塞式 IO 通常在代码进行 read( ) 调用时,代码会阻塞直至有可供读取的数据。同样,write( ) 调用将会阻塞直至数据能够写入。NIO 调用不会阻塞。相反,您将注册对特定 I/O 事件的兴趣 – 可读的数据的到达、新的套接字连接等等,而在发生这样的事件时,系统将会告诉您。使用异步 I/O 可以监听任何数量的通道上的事件,不用轮询,也不用额外的线程。

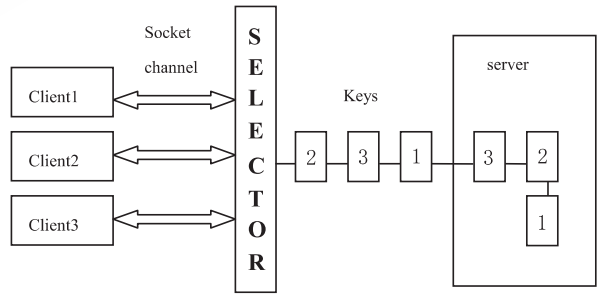

图 5　非阻塞通信体系结构

非阻塞通信服务器应用开发的步骤:

(1) 创建服务器的套接字通道。ServerSocketChannel

ssc = ServerSocketChannel. open( ) ;

(2) 设置非阻塞的 IO 工作方式。ssc. configureBlocking (false) ;

(3) 将连接绑定到端口。

ServerSocket ss = ssc. socket( ) ;

InetSocketAddress address = new InetSocketAddress(port) ;

ss. bind(address) ;

(4) 创建服务器端选择器对象。Selector selector = Selector. open( ) ;

(5) 将新打开的通道 ssc 注册到 selector 上

SelectionKey key = ssc. register(selector, SelectionKey. OP_ACCEPT) ;

Set selectedKeys = selector. selectedKeys( ) ;

Iterator it = selectedKeys. iterator( ) ;

(6) 进入无限循环,取得选择器对象的每个 Key。

SelectionKey key = (SelectionKey) it. next( ) ;

(7) 从 Key 集合中移除当前 Key。it. remove( ) ;

(8) 给取得的当前 Key 建立一个 Socketchannel 通道。

SocketChannel sc = (SocketChannel) key. channel ( )

(9) 通过此通道处理数据。

. . . . . .

4 简单应用实例

我们分别用基于阻塞与非阻塞 I/O 网络模型来建立一个服务器(echo server),它接受网络连接、监听端口并同时处理来自客户端所有的连接并向客户端简单的回显它们发送的数据。

4.1 基于阻塞 I/O 网络模型服务端的部分重要代码:

```
ServerSocket serverSocket = null; //声明一个 serverSocket
boolean listening = true; //声明一个监听标识
    serverSocket = new ServerSocket(1111); //初始化监听端口为 1111
        . . . . . .
    while( listening) //如果处于监听态则开启一个线程
    { //实例化一个服务端的 socket 与请求 socket 建立连接
        new
    EchoMultiServerThread ( serverSocket. accept ( ) ) .
start( ) ;
    } //将 serverSocket 的关闭操作放在循环外,只有当监听为 false 是,服务才关闭
        serverSocket. close( ) ;
    }
//多线程类的实现
public class EchoMultiServerThread extends Thread {
    . . . . . .
    public void run( ) {
        PrintWriter out = null;
        BufferedReader in = null;
        out = new PrintWriter( socket. getOutputStream ( ) , true) ;
    in = new BufferedReader( new InputStreamReader ( socket. getInputStream( ) ) ) ;
            out. flush( ) ;
        while( true) //只有当有用户输入的时候才返回数据
        {
        String str = in. readLine( ) ;
                . . . . . .
            out. println( str) ;
        out. flush( ) ;
                . . . . . .
    out. close( ) ;//收尾工作
    in. close( ) ;
    socket. close( ) ;
    . . . . . .
```

在这种方式下,服务器可以并发处理多个请求,但是随着同时连接的客户端数量的增加,系统中将存在大量的用于同客户端进行交互的工作线程。由于每个线程都有自己的栈空间,在不同线程间切换也是比较费时的操作,因此在这种模式下随着同时连接的客户端数量的增加,系统性能将下降得非常快。

4.2 基于非阻塞 I/O 网络模型服务端的部分重要代码:

首先建立一个缓冲区 echoBuffer:

```
private ByteBuffer echoBuffer = ByteBuffer. allocate (
```

```
1024 );
```
接着创建一个 Selector:
```
Selector selector = Selector.open();
```
打开一个 ServerSocketChannel **并设置为非阻塞的,将它绑定到给定的端口。**
```
ServerSocketChannel ssc = ServerSocketChannel.open();
ssc.configureBlocking( false );
ServerSocket ss = ssc.socket();
InetSocketAddress address = new InetSocketAddress( port );
ss.bind( address );
```
将新打开的 ServerSocketChannel **注册到** selector **上。**
```
SelectionKey key = ssc.register( selector, SelectionKey.OP_ACCEPT );
```
下面将进入主循环。

首先调用 Selector **的** select() **方法。这个方法会阻塞,直到至少有一个已注册的事件发生。当一个或者更多的事件发生时,** select() **方法将返回所发生的事件的数量。**
```
int num = selector.select();
```
接下来调用 Selector **的** selectedKeys() **方法,它返回发生了事件的** SelectionKey **对象的一个集合 。通过迭代** SelectionKeys **并依次处理每个** SelectionKey **来处理事件。**
```
Set selectedKeys = selector.selectedKeys();
Iterator it = selectedKeys.iterator();//建立迭代器
    while (it.hasNext()) {
       SelectionKey nkey = (SelectionKey)it.next();//取得相应事件的键
ServerSocketChannel ssc2 = (ServerSocketChannel) nkey.channel();
            //给取得的当前 Key 建立一个 Socketchannel 通道
          SocketChannel sc = ssc2.accept();
          int bytesEchoed = 0;
          while (true) {// 回应数据
            echoBuffer.clear();//清空缓冲区
            int r = sc.read( echoBuffer ); //把通道中的数据读至缓冲区
            if (r <=0) {
              break;
            }
            echoBuffer.flip();//缓冲区重置,为写做准备
            sc.write( echoBuffer ); //把缓冲区中的数据写至通道
                .....
                  it.remove();//删除处理过的 SelectionKey
                .....
```

5 结束语

传统的阻塞通信适用于同步通信及错误控制,而随着对 Java NIO **研究的深入,它的优点更加突出:第一,线程不再在通信时阻塞,第二,**Selector **能够处理多个连接,从而大幅降低了服务器应用程序开销。依靠非阻塞套接字技术,我们可以不用手工来处理多线程。很多网络邮件系统。异步通信(如** IRC **和聊天服务)最好使用非阻塞通信来获得更高的并发联接数、更小的系统开销和更好的性能。**

参考文献

1 萨维茨(Savitch,W.)Java 语言计算机科学与程序设计.北京:清华大学出版社,2005.

2 殷兆麟 Java 网络编程.北京:国防工业出版社,2001.

3 飞思科技产品研发中心 Java TCP/IP 应用开发详解.北京:电子工业出版社,2002.

4 结城浩(日本)博硕文化 译 Java 多线程设计模式.北京:中国铁道出版社,2005.

5 汪晓平 俞俊 李功 Java 网络编程.北京:清华大学出版社,2005.